Summer 7-31-2015

FUNDAMENTAL PROBLEMS IN POROUS MATERIALS: EXPERIMENTS & COMPUTER SIMULATION

Zhanping Xu
University of Nebraska-Lincoln, zhanpingxu@gmail.com

FUNDAMENTAL PROBLEMS IN POROUS MATERIALS: EXPERIMENTS & COMPUTER SIMULATION

by

Zhanping Xu

A Dissertation

Presented to the Faculty of

The Graduate College at University of Nebraska

In Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy

Major: Engineering (Materials Engineering)

Under the Supervision of Professor Li Tan

Lincoln, Nebraska

August, 2015

FUNDAMENTAL PROBLEMS IN POROUS MATERIALS: EXPERIMENTS & COMPUTER SIMULATION

Zhanping Xu, Ph.D.

University of Nebraska, 2015

Advisor: Li Tan

Porous materials have attracted massive scientific and technological interest because of their extremely high surface-to-volume ratio, molecular tunability in construction, and surface-based applications. Through my PhD work, porous materials were engineered to meet the design in selective binding, self-healing, and energy damping. For example, crystalline MOFs with pore size spanning from a few angstroms to a couple of nanometers were chemically engineered to show 120 times more efficiency in binding of large molecules. In addition, we found building blocks released from those crystals can be further patched back through a healing process at ambient and low temperatures down to -56 ºC. When building blocks are replaced with graphenes, ultra-flyweight aerogels with pore size larger than 100 nm were made to delay shock waves. More stable rigid porous metal with larger pores (~um) was also fabricated, and its performance and survivability are under investigation. Aside from experimental studies, we also successfully applied numerical simulations to study the mutual interaction between the nonplanar liquid-solid interface and colloidal particles during the freezing of the colloidal suspensions. Colloidal particles can be either rejected or engulfed by the evolving interface depending on the freezing speed and strength of interface-particle interaction. Our interactive simulation was achieved by programming both simulation module and visualization module on high performance GPU devices.

ACKNOWLEDGEMENTS

I am very grateful to my advisor, Dr. Li Tan for his inspiration and continuous help throughout my graduate studies at the University of Nebraska-Lincoln (UNL) and the writing of this dissertation. Dr. Tan always encouraged me to think creatively on experiment designs, rather than just do trial-and-error-type research. Those designs, some of which turned out to be not good, are all excellent training experience in creating, analyzing, and solving research problems. Special acknowledgement goes to Dr. Hongfeng Yu of the Department of Computer Science, for his kind and selfless support on the GPU programming, and Dr. Xiaocheng Zeng of the Department of Chemistry, advisor for my master degree, for his continuous coaching in simulation and modeling even years after I graduated from his lab. I also thank the other members of supervisory committee, Dr. Joseph A. Turner of the Department of Mechanical and Materials Engineering, Dr. Jian Zhang of the Department of Chemistry, and Dr. Laurent Delbreilh of the Université de Rouen, for their input and evaluation of the dissertation.

I would like to show my gratitude to Dr. Ziguang Chen and Dr. Gonghua Wang, former graduate student and postdoctoral researcher in Dr. Li Tan's group, for the pleasant cooperation in several projects. Those research works could not be published without their efforts. In Dr. Tan's lab, I also overlapped with and learned from: Dr. Jinyue Jiang, Dr. Alexandre Dhotel, Shumin Li, Rachel Horzewski, and Joseph Beeson.

Finally I dedicate this dissertation to my wife, Shenru Gao, my mother, Lanxiu Huang, my father, Ruxin Xu, my brothers Zhan and Zhanhui, and my sister Zhanmei for their greatest love, help and the trust they put in me.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Porous materials have attracted massive scientific and technological interest over decades because their extremely high surface-to-volume ratio. Atoms, ions and molecules can interact with the porous materials not only on the outer surface, but also inside the pores. Therefore, porous materials have been traditionally used in gas adsorption (separation), catalysis, ion exchange, and many of these applications have achieved commercialized production.

The pore size, which is associated with the transport mechanisms, plays a critical role for most applications. Table 1.1 shows the three pore size regimes defined by IUPAC[1.1]: micropores, which have diameter smaller than 2 nm; mesopores, which have diameters between 2 and 50 nm; macropores, which have diameters larger than 50 nm. The transport mechanisms can be very different for different pore size. Micropores are comparable to the size of molecules, so activated transport is the dominating mechanism. Mesopores are in the same order or smaller than the mean free path length. In this case, Knudsen diffusion and surface diffusion dominates. Macropores are larger than the typical mean free path length of typical fluid. Bulk diffusion is the main transport mechanism, and viscous flow also happens in a pressure environment.

Table 1.1 Three pore size regimes defined by IUPAC

pore regime	micropore	mesopore	Macropore
pore diameter d	$d < 2$ nm	$2 < d < 50$ nm	$d > 50$ nm
dominant transport mechanism	activated transport	Knudsen diffusion, surface diffusion capillary condensation	bulk diffusion, viscous flow

In addition to pore space, the type of building blocks (atoms, molecules, particles) and the way they form the solid in porous materials are also very important. If building blocks are assembled in a well-defined order via chemical bonding, the crystalline porous materials with

uniform pore size can be developed, e.g. Zeolite, aluminophosphates ($AlPO_4$), Metal-Organic Frameworks (MOFs), and Covalent-Organic Frameworks (COFs). The pore sizes of those materials usually fall in the range of micropores and lower-end of mesopores. On the other hand, non-crystalline porous materials with relatively less uniform pore sizes are obtained if building blocks are connected randomly via van der Waals interaction, including silica gels, particle-based assemblies, and free-dried samples. The pore sizes of those materials are usually in the range of macropores and higher-end of mesopores.

Many processes have been developed to prepare non-crystalline porous materials, including phase separation[1.2, 1.3], particulate leaching[1.4], 3D printing[1.5], and freeze-drying[1.6]. These methods, along with some new designs, have been employed to fabricate novel porous materials and new applications have been explored. Among one of these, freeze-drying can produce a large variety of porous materials depending on the freezing conditions. For instance, Sun H, Xu Z, and Gao C fabricated all carbon multifunctional Ultra-Flyweight Aerogels (UFAs) with density down to 0.16 mg cm^{-3} (density of air at ambient condition is 1.2 mg cm^{-3}) by freeze-drying aqueous solutions of CNTs and giant graphene oxide (GGO) followed by chemical reduction of GGO into graphene (Figure 1.1 A and B).[1.7] The UFA framework is constructed with giant graphene sheets serving as cell walls and CNTs as ribs (Figure 1.1 C), which yields integrated properties, such as excellent elasticity, ultralow density, outstanding thermal stability, and extremely high absorption capacities for organic solvent. The UFA with density 0.75 mg/cm^3 was reported to be highly hydrophobic with an average pore size of 123 nm, which leading to an extremely high absorption rate of 68.8 g g^{-1} s^{-1} for toluene, and absorption capacities of 215-843 g g^{-1} depending on the liquid density (Figure 1.1 D). The extremely high

absorption capacity and absorption rate are resulted from the high surface tension between the organic solvent and graphene sheets.

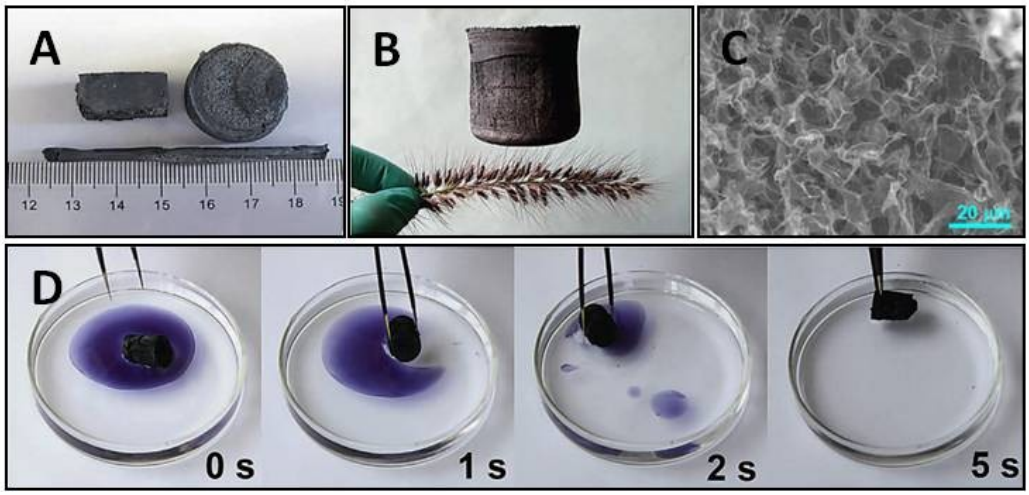


Figure 1.1 Macroscopic and Microscopic structures of UFAs. A) and B) free standing UFAs. C) The UFA framework is constructed with giant graphene sheets serving as cell walls and CNTs as ribs. D) Absorption process of toluene (stained with Sudan Black B) on water by the UFA within 5 s. (Courtesy of Ref 1.7)

Another good example of important role of surface tension in porous materials was carried by Qiao's group. Recently, they demonstrated that hydrophobic porous silica gel (average pore size 100 nm) can serve as an energy capture media for mitigation of intense stress waves, by performing molecular simulations and split Hopkinson Bar experiment.[1.8] The energy capture is a non-dissipative energy absorption mechanism. The liquid overcomes the capillary effect and infiltrates into the nanopores under the pressure of the stress wave. The mechanical energy is temporarily stored by the confined liquid and isolated from the wave energy transmission path during this process, which greatly reduces the pressure and the energy of the transmitted stress wave (Figure 1.2). The captured energy is gradually released after the impact process ends.

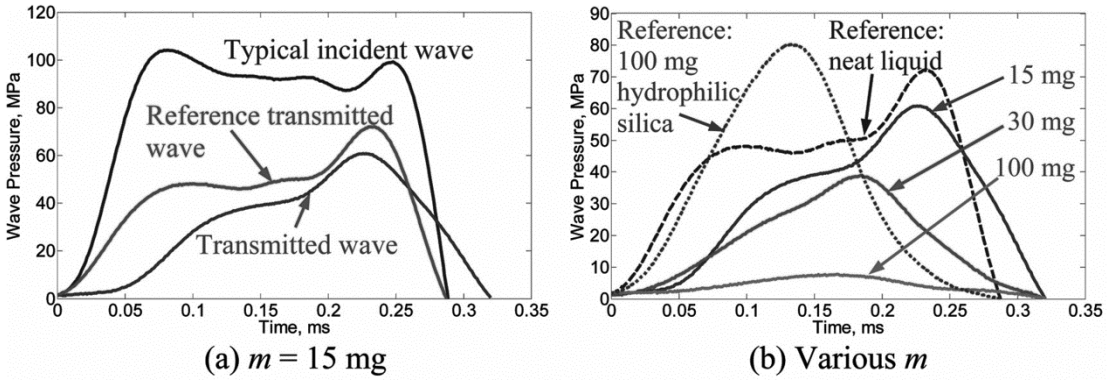


Figure 1.2 The energy capture effect of a nanoporous silica gel immersed in a nonwetting liquid: (a) The incident and the transmitted waves of a sample containing 15 mg of hydrophobic silica gel; the reference curve is the transmitted wave of neat liquid. (b) The transmitted waves of samples of different amounts of hydrophobic silica gel. (Courtesy of Ref 1.8)

In the above two cases, the porous solid framework is chemically inert to the applied solvent and surface energy is the main interaction. However, porous MOFs, in which the open framework crystalline structures are constructed by coordination of metal ions to organic "linker" moieties, are chemically active. They have been demonstrated to interact with the environment through non-covalent bonding. For example, Kim M, Cahill JF, Fei H, .etc have shown that postsynthetic ligand and metal ion exchange (PSE) processes occur even in most "inert" MOFs.[1.9] Their results strongly suggest that PSE is nearly universal in MOFs and that the chemical bond between the metal cluster secondary building units (SBUs) and ligand linkers is reversible. While this chemical instability makes MOFs vulnerable in some cases, the dynamic properties open the possibility of their use in dynamic combinatorial chemical systems that cannot be achieved with prior amorphous but porous counterparts. In addition, the selective sorption of certain guest molecules promotes MOFs as potential candidates for sensing and separation of the guest molecules. The selectivity was reported not only resulted from the size and geometry confinement, but also from the host-guest interactions.[1.10]

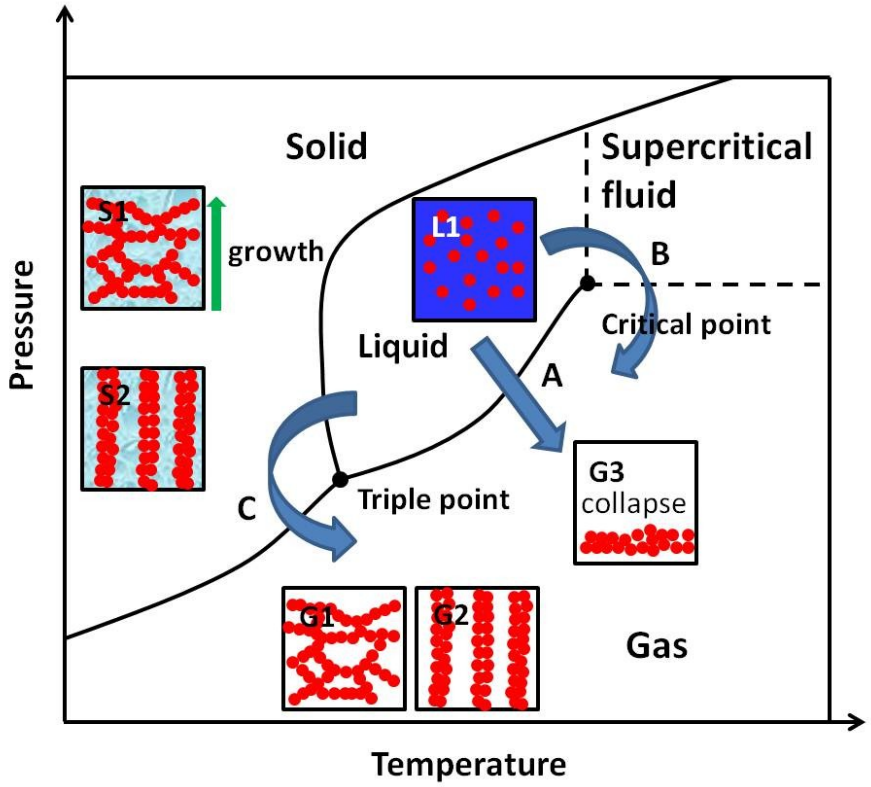


Figure 1.3 Phase diagram for a colloid suspension. If the colloid suspension is dried by direct removal of solvent (via evaporation or heat), no free standing structure can be obtained because of pulling from surface tension (route A, inlet image G3). In order to avoid this, two possible alternate paths can be used to remove the solvent without crossing the liquid-gas boundary. In supercritical drying (route B, going right, high-temperature, high-pressure side), the transition from liquid to gas does not across any phase boundary, instead passing through the supercritical region, where distinction between liquid and gas disappears. The other route is freeze-drying (route C, going left, low-temperature, low-pressure side), which the solution is frozen in a cold bath, followed by solvent removal via sublimation at low pressure, leading to formation of various porous structure of solute phase (e.g. inlet images S1 & G1 – network structure, inlet images S2 & G2 – fibers). The morphologies of the solute phase are determined by the freezing temperature, the nature of the solute and solvent, and the freezing direction.

As the structure and porosity of amorphous porous materials are strongly affected by the process of fabrications, we use Figure 1.3 to illustrate the phase diagram for a colloid suspension

and possible routes for fabrication of porous structure. If the colloid suspension is dried by direct removal of solvent (via evaporation or heat), no free standing structure can be obtained because of pulling from surface tension (route A, inlet image G3). In order to avoid this, two possible alternate paths can be used to remove the solvent without crossing the liquid-gas boundary. In supercritical drying (route B, going right, high-temperature, high-pressure side), the transition from liquid to gas does not across any phase boundary, instead passing through the supercritical region, where distinction between liquid and gas disappears. Carbon dioxide (critical point 304.25 K at 7.39 MPa) is a commonly used fluid for supercritical drying, while water is usually inconvenient for this purpose (647 K, 22.064 MPa) due to the possible heat damage to the sample. The other route is freeze-drying (route C, going left, low-temperature, low-pressure side), which the solution is frozen in a cold bath, followed by solvent removal via sublimation at low pressure, leading to formation of various porous structure of solute phase (e.g. inlet images S1 & G1 – network structure, inlet images S2 & G2 – fibers). The morphologies of the solute phase are determined by the freezing temperature, the nature of the solute and solvent, and the freezing direction. However, the freezing process is very dynamic and far from well-understood. A number of experimental and theoretical studies have been carried to understand the solidification of colloidal suspensions under unidirectional thermal gradient. Among them, Suzuki Y, Sazaki Gen, Hashimoto K, .etc designed a unidirectional freezing stage to observe in situ the freezing process of the colloidal suspensions.[1.11] Their results indicate the particle moving and colloidal crystallization in front of the growing ice-water interfaces. Peppin SSL[1.12] developed a mathematical model to describe the unidirectional solidification of a suspension of hard-sphere particles. Volume fraction and temperature profiles ahead of a planar solidification front were obtained. Garvin JW and Udaykumar HS performed numerical simulations to study the

interaction between a solidification front with an embedded particle.[1.13, 1.14] The critical velocity (below which the particle is engulfed otherwise pushed away by the solidification interface) was found to be greatly affect by the expression of drag force used in their model.

Through my PhD study, different porous materials were engineered to satisfy the requirements for some novel applications. Besides, numerical models were also developed to simulate the in-situ growing dynamics in porous material processing. The dissertation is organized as follow. In Chapter 2, chemically engineered porous Cu-MOFs are demonstrated to enhance the binding of large molecules, as well as the capability of sequential binding. Chapter 3 shows that the building blocks released from porous MOFs can be further patched back through a healing process at ambient and low-temperature conditions. In Chapter 4, ultra-flyweight aerogels (UFAs) and related carbon based structured materials are fabricated via freeze-drying. Stable rigid porous copper thin films with micrometer pores were also made by colloidal templating and electrodeposition. Their applications in energy delay/mitigation of destructive shock wave are explored. In Chapter 5, a multiscale numerical model is developed to simulate the growth dynamics in controlled-freezing of colloidal suspensions. In-situ visualization of the growth process is achieved by coupling the visualization module with simulation module on high performance GPU devices. Lastly research work through my PhD are summarized in Chapter 6. Existing issues and future research directions are also listed.

References:

[1.1] U. Schubert, N. Husing, *"Synthesis of Inorganic Materials"*, John Wiley & Sons, 2012.

[1.2] C.Schugens, C. Maquet, C. Grandfils, R. Jerome, and P. Teyssie, *Polymer* **37**, 1027–1038 (1996).

[1.3] C. Schugens, V. Maquet, C. Grandfils, R. Jeromem, and P. Teyssie, *J Biomed Mater Res* **30**, 449–461 (1996).

[1.4] A. G. Mikos, A. J. Thorsen, L. A. Czerwonka, Y. Bao, R. Langer, and D. N. Winslow, *Polymer* **35**, 1068–1077 (1994)

[1.5] S. J. Hollister, *Nat. Mater.* **4**, 518-524 (2005)

[1.6] H. W. Kang, Y. Tobata, Y. Ikada, *Biomaterials* **20**, 1339-1344 (1999).

[1.7] H. Sun, Z. Xu, and C. Gao, *Adv. Mater.* **25**, 2554-2560 (2013).

[1.8] B. Xu, X. Chen, W. Lu, C. Zhaom, and Y. Qiao, *Appl. Phys. Lett.* **104**, 203107 (2014).

[1.9] M. Kim, J. F. Cahill, H. Fei, K. A. Prather and S. M. Cohen, *J. Am. Chem. Soc.* **134**, 18082–18088 (2012).

[1.10] M.P. Suh, Y.E. Cheon, E.Y. Lee, *Coord. Chem. Rev.* **252**, 1007 (2008).

[1.11] Y. Suzuki, G. Sazaki, K. Hashimoto, T. Fujiwara, and Y. Furukawa, *J. Crystal Growth* **383**, 67 (2013).

[1.12] S. S. L. Peppin, J. A. W. Elliott, and M. G. Worster, *J. Fluid Mech.* **554**, 147-166 (2006).

[1.13] J. W. Garvin and H. S. Udaykumar, *J. Crystal Growth* **252**, 451 (2003).

[1.14] J.W. Garvin and H.S. Udaykumar, *J. Crystal Growth* **252**, 467 (2003).

CHAPTER 2

SEQUENTIAL BINDING OF LARGE MOLECULES TO HAIRY MOFS

2.1 Introduction

Metal–organic frameworks (MOFs) are a class of sophisticated crystalline solids with well-defined coordination geometry and porosity.[2.1-2.4] Rich selections of building blocks, long-range ordering in packing, and superior surface areas have promoted MOFs' applications in gas adsorption, storage, and separation.[2.5–2.10] While most studies utilized inner pores by regulating their interactions with small molecules, MOFs are rarely reported to bind large molecules, for instance, polymers[2..11–2.13] or proteins.[2.14, 2.15] Two major reasons are the limited pore sizes and chemical instability of the frameworks. Though expanded pores have been reported for hosting large molecules such as proteins,[2.16] most of the MOFs have pore size spanning from several angstroms to a couple of nanometers.[2.17] The size exclusion prevents large molecules from entering the inner pores, and hence they stick to the outer surfaces only.[2.18] Chemical instability makes MOFs vulnerable when in contact with liquid media such as water and organic solvent, through ligand exchange processes.[2.19–2.23] A combination of both factors therefore makes a controlled binding of large molecules challenging.

2.2 Materials and Method

2.2.1 Materials

In our experiments, $Cu(FMA)(4,4'\text{-}Bpe)_{0.5}\cdot 0.5H_2O$ (Cu-MOFs) were used, where FMA is fumarate ion ($C_4H_2O_4^{2-}$) and 4,4'-Bpe is 4,4'-vinylenedipyridine ($C_{12}H_{10}N_2$). Cu-MOFs were synthesized after an established "shake-and-bake" method.[2.24] Briefly, a mixed solution of $Cu(NO_3)_2\cdot 2.5H_2O$ (0.0663 g, 0.285 mmol), H_2FMA (0.0331 g, 0.285 mmol), and 4, 4'-Bpe (0.0260 g, 0.143 mmol) in H_2O (25 mL) in a 30 mL glass vial was heated in a convection oven

(DKN400, Yamato Scientific America, Inc.) at 100 °C for 24 h. The greenish Cu-MOF crystals (~ 40% yield) were obtained after a copious filtration, hot water rinsing, and air drying. Cu-MOF-1 was obtained by soaking Cu-MOF in an aqueous NaCl solution (57 mM) and then following filtration, repeated rinsing and air drying. $CuCl_2$·(4,4'-Bpe) was synthesized as an effort towards identifying hairy MOFs. Anhydrous $CuCl_2$ (0.148 g, 1.10 mmol) and 4, 4'-Bpe (0.100 g, 0.55 mmol) was mixed, added with 30 ml of deionized water, the mixture was heated in a convection oven at 100 °C for 24 h. The $CuCl_2$·(4,4'-Bpe) complex powders were obtained after filtration, rinsing and drying. CuFMA was synthesized according to an established method.[2.25] In a typical synthesis, $CuCO_3$·$Cu(OH)_2$ (0.0144 g, 0.0651 mmol) and H_2FMA (0.0758 g, 0.653 mmol) was added to 10 ml of deionized water, heated in a convection oven at 100 °C for 24 h. Light blue CuFMA powders were collected after filtration, rinsing and drying process.

Protein adhesion tests were performed using GFPs (GFP_{uv} with excitation wavelength of 395 nm and emission wavelength of 510 nm) and RFPs (m-cherry with excitation wavelength of 587 nm and emission wavelength of 610 nm). Both proteins, stored in potassium phosphate buffer (50 mM, pH 7.4), were expressed with a hexahistidine tag, which can enhance their binding with metal sites. The interaction between GFPs and the copper compounds were probed by soaking the Cu compounds (Cu-MOF, Cu-MOF-1, $CuCl_2$·(4,4'-Bpe) and CuFMA) in an aqueous solution of GFPs (~3.0 μg/mL) for 1 week at 4 °C, followed by examination using confocal fluorescence microscope. The controlled salt etching process was revealed by alternate RFPs and GFPs staining. The CuMOF crystals were first etched with NaCl solution (57 mM) for 1 day, after removing the solution, the solid was soaked in RFP solution for 1 week, followed by a second NaCl etching, washing and soaking in GFPs solution. Dulbecco's Modified Eagle

Medium (DMEM) with added fetal bovine serum (FBS) was prepared according to manufacturer's instruction (Life Technologies, Carlsbad, CA).

2.2.2 Characterizations

X-ray diffractometry (XRD, Bruker AXS D8 Discover with GADDS) was conducted to examine the crystal structures of Cu-MOF, Cu-MOF-1, $CuCl_2$·(4,4'-Bpe) and CuFMA samples. The weighted average wavelength of the Cu Kα x-ray source was 1.5417 Å. Atomic force microscopy (AFM, Dimension 3100 SPM system) was performed to reveal the surface morphology of the Cu-MOFs, crushed Cu-MOFs and its morphological developments in salt solutions with various concentrations. Before AFM imaging, the Cu-MOF crystals were soaked in the salt solutions for 10 minutes before washing and drying. The Scanning Probe Image Processor software (SPIP, Image Metrology, Denmark) was used to analyze the surface roughness and depth profiles. Optical images of the Cu-MOF were taken with an optical microscope (Meiji ML8000) equipped with a digital camara (Moticam 2000). The GFPs-containing samples were examined with a 60× (PlanApo/1.45) oil lens and imaged with an Olympus FV500-IX81 confocal laser scanning microscope system using the 488 nm excitation laser line (Olympus America Inc., Center Valley, PA). For the samples with both GFPs and RFPs, images were collected using the sequential mode using 488 nm/543 nm excitations and 520 nm/595 nm emissions, respectively. Elemental analysis of $CuCl_2$·(4,4'-Bpe) was performed in Atlantic Microlab, Inc. (Norcross, GA).

2.3 Results and Discussion

2.3.1 Trapping of Large Molecules on Surface Defects

We first found that defects like cracks or edges in MOFs are effective traps for retaining large molecules like proteins (Figure 2.1A). A simple mechanical crushing reveals more active sites and greatly enhances protein adsorption onto particular crystal planes (Figure 2.1B–E).

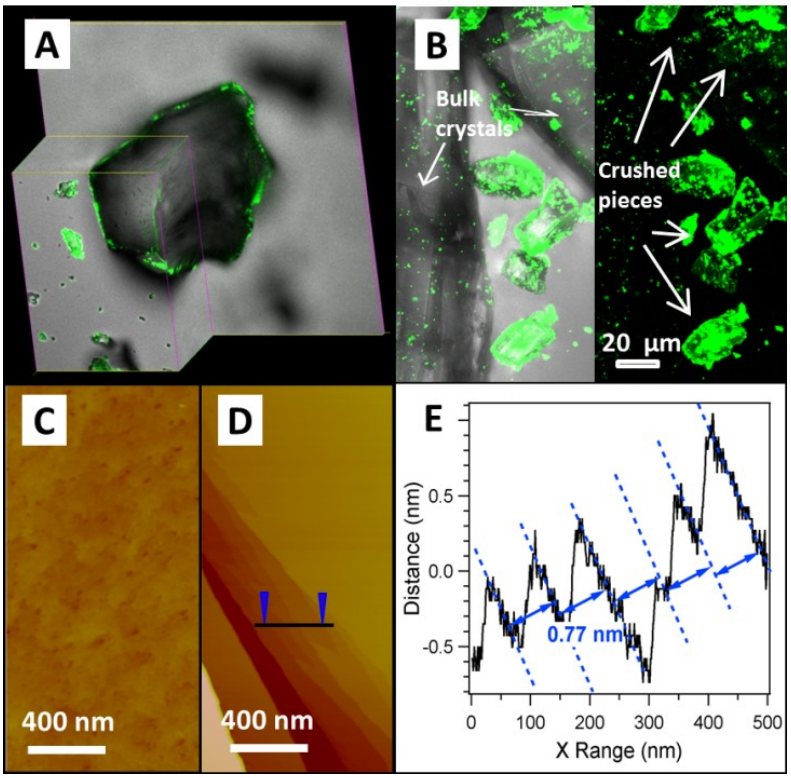


Figure 2.1 Confocal fluorescent microscopic images of protein (GFPs) adhesion on (A) bulk and (B) crushed Cu-MOF crystals; AFM images of (C) the bulk and (D) crushed MOF crystal; (E) Section analysis of (D) along the line revealing the surface terraces with a fringe spacing of 0.77 nm, matching crystal planes of (200)s.

Table 2.1 Major planes in Cu-MOFs and their anisotropy in composition and strength.

Experimental XRD (2θ) Position	Simulated XRD (2θ) Position	Crystal Plane	Area (A^2)	No. of Inter-layer Bonds (Cu-O, Cu-N)	Interlayer Strength ($kJ/N_A \cdot A^2$)	In-plane Strength ($kJ/N_A \cdot A^2$)
11.175º	11.174º	(2 0 0)	146.18	(0, 2)	1.23	18.28
12.687º	12.672º	(1 1 1)	167.28	(4, 2)	5.07	7.99
13.610º	13.601º	(0 0 2)	160.15	(8, 0)	8.34	2.25
16.011º	16.011º	(0 2 0)	214.53	(8, 0)	6.23	1.68
21.066º	21.066º	(0 2 2)	277.44	(4, 0)	2.41	6.11

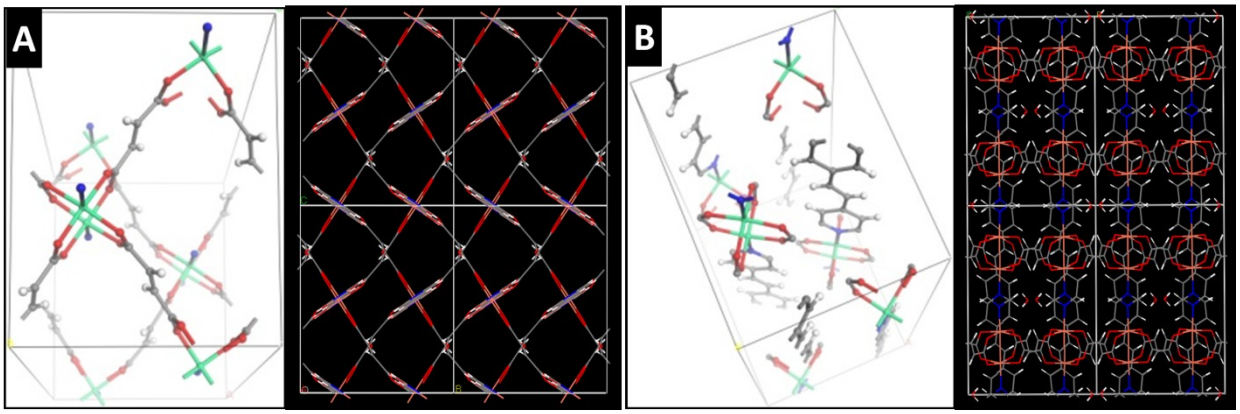


Figure 2.2 Anisotropic structure of Cu-MOFs. (A) Selected view of the (200)s in a box, with its plane view on the right. Each knots (red cross) is composed of 2 copper atoms (in green) that are coordinated with 8 oxygen atoms (in red) from the acid (in gray) to form a continuous network; and (B) simplified (002)s in a box, with its plane view on the right. Each pillar in the box is composed of nitrogen atoms (in blue) that are coordinated with copper from the top or bottom.

The Cu-MOFs are assembled in an interpenetrating pseudo-cubic crystal structure, with the (200) layers composed of pure Cu-O bonds, and pillar-like 4,4'-Bpe units between the (200) layers (Table 2.1, Figure 2.2). Since the binding energy of Cu-N bond (90 kJ/mole) is much lower than that of Cu-O bond (186 kJ/mole), the (200) planes have a smaller inter-plane strength. Essentially, the inter-planar pillar linkers are weaker than the in-plane carboxylate moieties, therefore when mechanically crushed, the fragile Cu-N bonds that connect the (200) planes are easier to break, likely leaving dangling 4,4'-Bpe and coordinatively unsaturated Cu atoms.

2.3.2 Fabrication of Hairy MOFs (H-MOFs) via Salt-etching and Characterization

As mentioned in previous section, Cu-MOFs are assembled in an interpenetrating pseudo-cubic crystal structure, with pure Cu–O bonded (200) layers and pillar-like units between the layers. The etching process was performed to break the (200) layers and we investigated the effects using an atomic force microscope (AFM). Prior to imaging, these MOFs were treated

with a series of salt solutions, e.g., 0.57, 5.7, 57 and 570 mM, respectively, for 10 minutes. The height images of the four crystal surfaces (Figure 2.3) revealed an enhanced surface roughening as the salt concentration increased. When the crystals were treated with the most diluted salt solution (0.57 mM), their smooth surfaces became densely bumpy ones, with the surface roughness increased from 0.82 to 3.02 nm (Figure 2.1C and 2.3A, Table 2.2). When the salt concentration increased by 10 times, discrete rectangular-shaped pit holes emerged (Figure 2B). As the salt concentration further increased by 100 times, many more pit holes appeared and some of them grew into larger ones, indicating possible migration of crystal grains (Figure 2.3C). When the crystals were soaked in a 570 mM salt solution, trenches appeared, further indicating possible surface or subsurface reconstructions (Figure 2.3D).

Table 2.2 Surface morphology evolution analysis based on Figure 2.3 and 2.4. The peak to peak distance measures the distance between the minimum peak depth and depth at histogram maximum. All data are obtained with the SPIP program.

Salt Concentration (mM)	Roughness (R_q, nm)	Peak to Peak Distance (nm)	Material Volume (μm^3)	Void Volume (μm^3)	Void Volume Fraction (%)
0.57	3.02	39.07	0.081	0.076	48
5.7	6.92	115.86	0.307	0.158	34.0
57	12.9	137.59	0.351	0.201	36.4
570	13.4	100.44	0.204	0.199	49.4

The statistical analysis of the peak to peak distances and void space volumes, based on the height histograms, also suggests that our etching process expedites with increasing salt concentration (Figure 2.4 and Table 2.2).

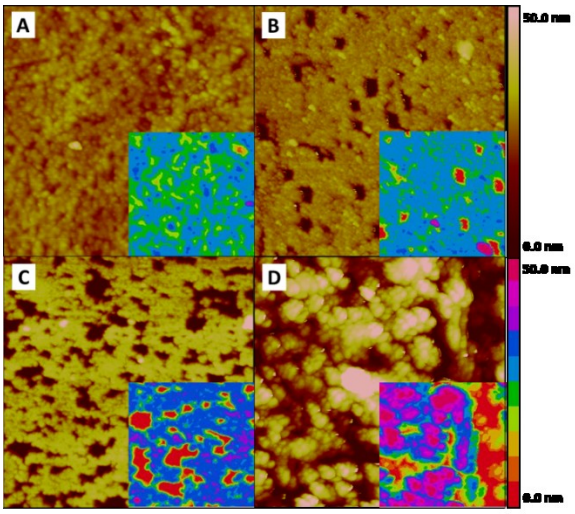


Figure 2.3 AFM images of Cu-MOFs treated by NaCl solutions with concentration of (A) 0.57, (B) 5.7, (C) 57 and (D) 570 mM. All images are 2 µm x 2 µm in size. The continuous and the sectional color scales are shown on the right side.

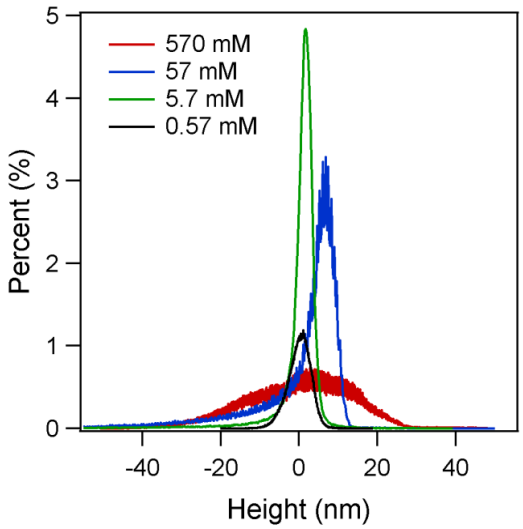


Figure 2.4 Height histogram of the AFM images (Figure 2.1) of the Cu-MOFs treated by NaCl solutions with concentration of 0.57, 5.7, 57 and 570 mM. The plot is based on the depth analysis data obtained with the SPIP program.

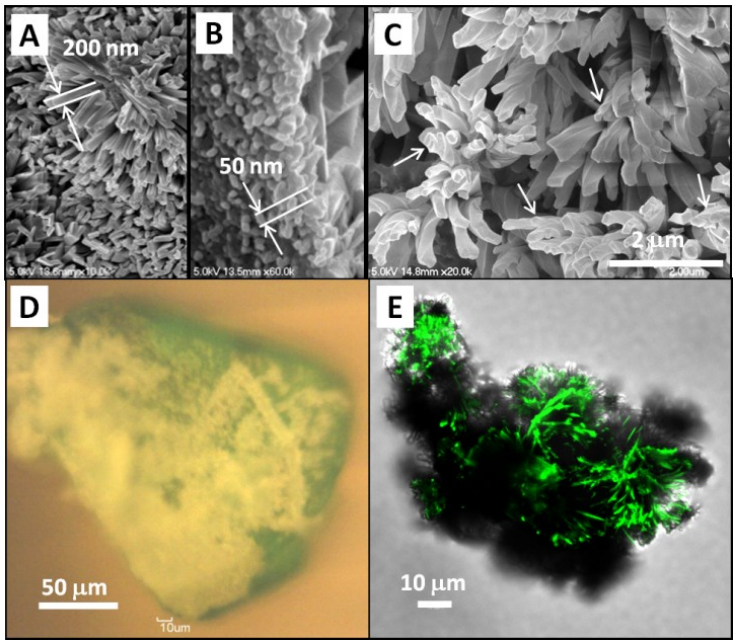


Figure 2.5 The morphological evolution of H-MOFs and enhanced surface GFP adhesion. SEM images of Cu-MOFs after soaking in a NaCl solution (57 mM) for (A) 30 min, (B) 24 hr. (C) roots of the fibers in (B) reveal 50 nm sized grains. (D) Optical microscopic image of the H-MOFs in (B). (E) Confocal fluorescent microscopic images of GFP-coated hairy MOFs.

Hairy MOFs show up whenever we impregnate the Cu-MOFs crystals in a salt solution (57 mM) for extended period of time, e.g., from 30 min to 24 hr. We found the morphology of the crystals continuously evolve as the etching proceeds (Figure 2.5). After a 30-min soaking, a large number of pit holes appear, consistent with our earlier AFM observations in Figure 2.3C. Right after 24 hrs, these crystals were covered with populated clusters of rods, with cross-section dimension in the range of 200-300 nm and lengths of approximately 15 μm (Figure 2.5A). A careful view of the hair roots reveals grain size of 50 nm (Figure 2.5B), suggesting reorganization of 4~5 of these into one larger rectangular hair. Often, this volume change in internal structure can twist the fibres into helical structures (Figure 2.5C). Overall, the shiny green crystals turned into opaque and furry under the lens of optical microscope (Figure 2.5D).

When the rod clusters, dubbed as hairy MOFs or H-MOFs, are placed in a GFPs solution, substantial fluorescence is captured, suggesting a strong affinity of H-MOFs to protein molecules (Figure 2.5E). In contrast to pristine Cu-MOFs, given the rod size in Figure 2.5B and a density of 10 rods/μm^2, salt-etching yields 120 times more contact area for molecule-binding.

While the interaction between hairy MOFs and GFPs provides hints of copper–protein binding, a challenge remains in identifying the structure of H-MOFs. We first probed the porous feature of H-MOFs by examining their gas sorption properties. The N_2 sorption isotherms, compared to the ones for Cu-MOFs,[2.10] show negligible N_2 sorption, indicating the breaking of the interpenetrating microporous framework of Cu-MOFs after salt etching (Figure 2.6). The CO_2 sorption isotherm of H-MOFs (Figure 2.6D) showing the sorption capacity about half of that absorbed by Cu-MOFs (Figure 2.6B), also suggests the cleaved framework structure of H-MOFs.

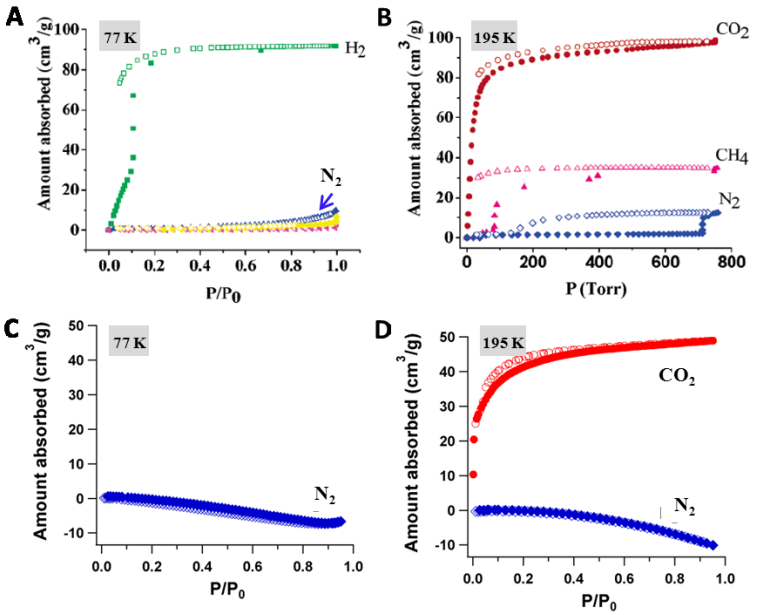


Figure 2.6 Gas sorption isotherms of Cu-MOFs and H-MOFs at 77 K and 195 K. (a) and (b) are gas sorption isotherms of Cu-MOFs adapted from Fig 2 in Ref.2. In (a), the H_2, N_2, Ar, and CO isotherms are shown in green, blue, magenta, and yellow.

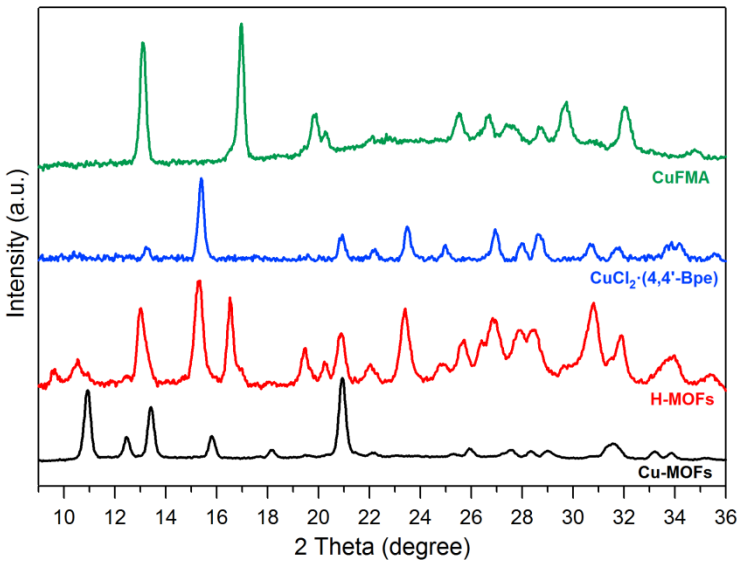


Figure 2.7 Powder XRD patterns of Cu-MOFs, H-MOFs, CuCl2·(4,4'-Bpe) and CuFMA. H-MOFs show the characteristic diffraction peaks of both CuCl2·(4,4'-Bpe) and CuFMA.

We further synthesized a series of crystalline materials containing different combinations of Cu, fumarate (FMA^{2-}) and 4,4'-Bpe. By comparing XRDs (Figure 2.7), H-MOFs held similar crystal structure with $CuCl_2$·(4,4'-Bpe), where elemental analysis yields a composition of C: 45.59%, H: 3.12%, N: 8.82%, and Cl: 22.54%. To be noted, there are a few extra XRD peaks at 16.5°, 19.4°, 20.3° and 25.7°, suggesting existence of $CuFMA^{2.22}$. Therefore, the hairy MOFs contain ingredients of both $CuCl_2$·(4,4'-Bpe) (major) and CuFMA (minor). We hypothesize that the salt etching has modified the (200)s in Cu-MOFs. Immediately after NaCl is introduced into the mother liquid of Cu-MOFs, the burst of Na^+ ions can disrupt the dynamic equilibrium of the MOF crystals and its mother liquid by depriving the fumarate moieties which are chelated to the copper centers in the (200) planes. As a consequence, the leaching of fumarate, which is the main building block of (200)s, unavoidably triggers the transformation of layered structures into fine fibers. Meanwhile, the Cu-N bonds adapt to these changes and reorganize the newly introduced species, namely chloride ions, to form the emerald-coloured $CuCl_2$·(4,4'-Bpe). Since

the entire etching process is handled at room temperatures, some CuFMA can precipitate out as a side-product.

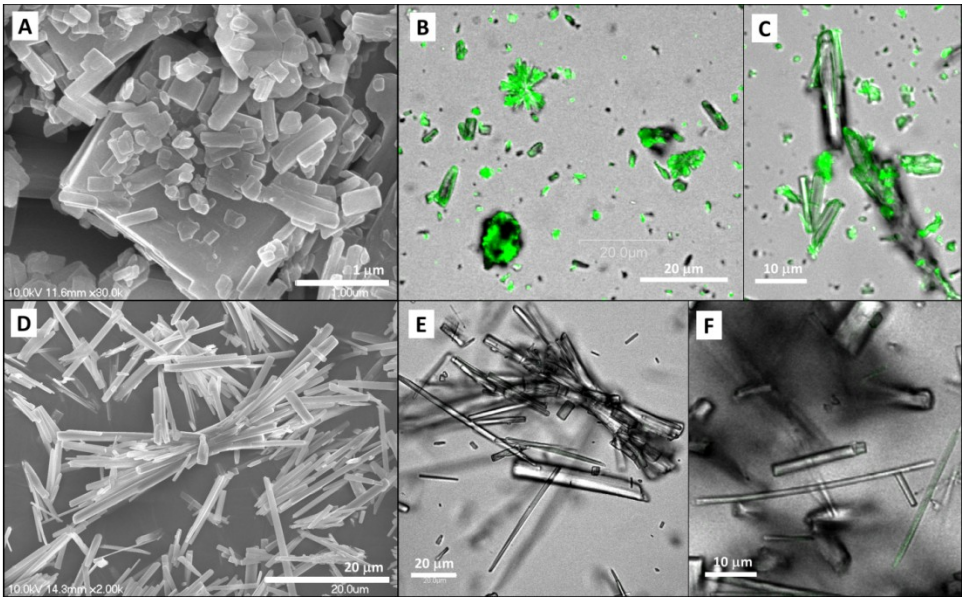


Figure 2.8 The morphologies of $CuCl_2$·(4,4'-Bpe) and CuFMA, and their interactions with GFP. (A) SEM image of $CuCl_2$·(4,4'-Bpe) showing rod-like structures coexisting with larger blocks; (B and C) Confocal fluorescent microscopic image of $CuCl_2$·(4,4'-Bpe) illustrating intense fluorescent response on the crystal surface, suggesting strong GFP adhesion; (D) SEM image of CuFMA showing uniform long rod-like morphology of the crystals and (E and F) Confocal fluorescent microscopic image of CuFMA with no detectable fluorescent signals, suggesting CuFMA is not active in binding GFP.

Moreover, we evaluated the increased active copper sites by comparing fluorescent response from $CuCl_2$·(4,4'-Bpe) and CuFMA respectively. Though both materials exhibit rod-like structures (Figure 2.8A and 2.8D), the confocal microscopic image of the $CuCl_2$·(4,4'-Bpe) in GFPs solution reveals bright fluorescent signals all through the solids (Figure 2.8B and 2.8C), while the CuFMA rods show hardly any fluorescent response (Figure 2.8E and 2.8F). This comparison clearly indicates that the $CuCl_2$·(4,4'-Bpe) in H-MOFs is indeed the compound that is active in binding GFPs, not CuFMA. We further extended the salt etching protocol with

common biological media. In parallel, we compared the behaviors of Cu-MOF and hairy MOFs in Dulbecco's Modified Eagle Medium (DMEM) with added fetal bovine serum (FBS) and GFPs. Unsurprisingly, site-selective fluorescent signals are found in this alternative experiment, while strong affinity of FBS on copper-rich hairy MOFs confirms intense and uniform adhesion of GFPs (Figure 2.9).

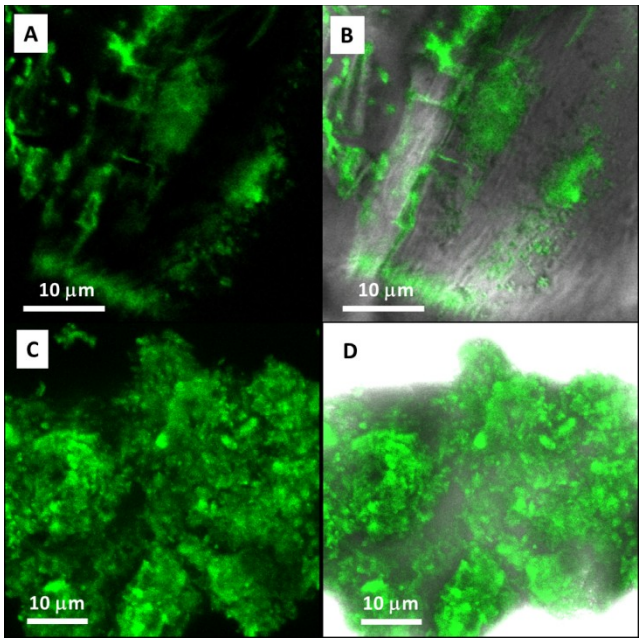


Figure 2.9 Confocal fluorescent microscopic images of Cu-MOFs and H-MOFs after soaking in FBS-containing DMEM and GFP for 1 week. (A and B) Fluorescent images of treated Cu-MOF unambiguously suggest the site-selective adhesion of GFPs on the defect sites of Cu-MOFs surface; (C and D) Fluorescent images of treated Cu-MOF-1 showing a uniform coverage of GFPs over the surface of H-MOFs, indicating the effectiveness of salt etching in enhancing bio-conjugation of Cu-MOF.

2.3.3 Sequential Binding of Large Molecules to H-MOFs via Stepwise Salt-etching

We applied this etching process to bind large molecules in a stepwise manner (Figure 2.10). Particularly, we used RFPs and GFPs alternately after individual etching process. After one day of reaction with NaCl solution, rod-like features grew on hairy MOFs and it showed a

strong interaction with RFPs, as in Figure 2.11A. After the removal of unbound RFPs and another round of salt etching, a new protein, i.e., GFPs, was used to decorate the solids. Correspondingly, the confocal fluorescent microscopic image show added but distinguishable green fluorescent signals (Figure 2.11B). If we compare these with the fresh $CuCl_2$·(4,4'-Bpe) rods which contain rather fixed surface copper sites, the H-MOFs are able to expose new sites after binding. In addition, the site-selective sequential binding results present a different binding motif from the hydrophobic-hydrophobic interaction reported in the MOFs with large pore sizes[2.16].

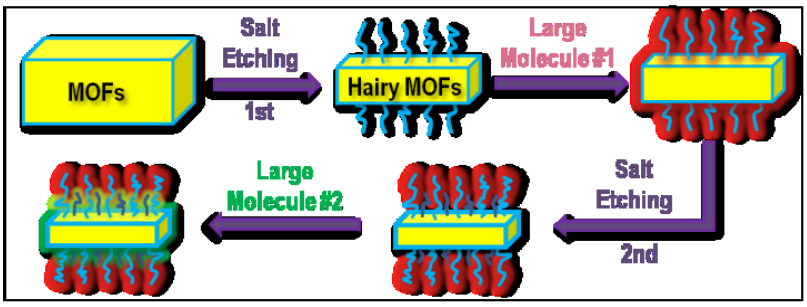


Figure 2.10 Schematic illustrations of the salt etching and sequential bindings of large molecules.

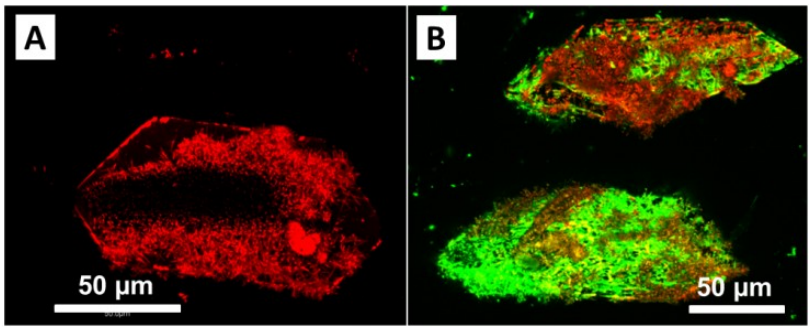


Figure 2.11 Confocal fluorescent microscopic images of H-MOFs (A) after soaking in RFP solution for 1 week and (B) after a second soaking in NaCl solution and followed by soaking in GFPs solution.

2.4 Summary

In summary, we have chemically engineered Cu-MOFs towards enriched Cu binding sites with a touch of salt. Analogous to the expedited rusting of irons in sea water, salty water can also accelerate the "corrosion" of crystalline Cu-MOFs. The salt solutions promoted the solvation and extraction of fumaric species from the original framework and delivered highly dense rod-like structures. Engineered hairy MOFs have demonstrated 120 times more enhanced binding of GFPs, as well as the capability of stepwise binding. Through this consecutive and selective binding, H-MOFs can serve as a microreactor for various biomolecular reactions. In addition, as metal-protein binding is a major contributor to the rigidity of electron transfers in many biochemical processes, we envision our work is beneficial to broad areas of biological research.[2.19]

References:

[2.1] N. Stock and S. Biswas, *Chem. Rev.* **112**, 933–969 (2012).

[2.2] H. C. Zhou, J. R. Long and O. M. Yaghi, *Chem. Rev.* **112**, 673–674 (2012).

[2.3] O. M. Yaghi, M. O'Keeffe, N. W. Ockwig, H. K. Chae, M. Eddaoudi and J. Kim, *Nature* **423**, 705–714 (2003).

[2.4] D. Zacher, R. Schmid, C. Woll and R. A. Fischer, *Angew. Chem., Int. Ed.* **50**, 176–199 (2011).

[2.5] R. E. Morris and P. S. Wheatley, *Angew. Chem., Int. Ed.* **47**, 4966–4981 (2008).

[2.6] J. R. Li, J. Sculley and H. C. Zhou, *Chem. Rev.* **112**, 869–932 (2012).

[2.7] M. P. Suh, H. J. Park, T. K. Prasad and D. W. Lim, *Chem. Rev.* **112**, 782–835 (2012).

[2.8] N. L. Rosi, J. Eckert, M. Eddaoudi, D. T. Vodak, J. Kim, M. O'Keeffe and O. M. Yaghi, *Science* **300**, 1127–1129 (2003).

[2.9] P. Horcajada, R. Gref, T. Baati, P. K. Allan, G. Maurin, P. Couvreur, G. Ferey, R. E. Morris and C. Serre, *Chem. Rev.* **112**, 1232–1268 (2012).

[2.10] B. L. Chen, S. Q. Ma, F. Zapata, F. R. Fronczek, E. B. Lobkovsky and H. C. Zhou, *Inorg. Chem.* **46**, 1233–1236 (2007).

[2.11] X. Q. Liang, F. Zhang, W. Feng, X. Q. Zou, C. J. Zhao, H. Na, C. Liu, F. X. Sun and G. S. Zhu, *Chem. Sci.* **4**, 983–992 (2003).

[2.12] M. D. Rowe, D. H. Thamm, S. L. Kraft and S. G. Boyes, *Biomacromolecules* **10**, 983–993 (2009).

[2.13] T. Ben, C. J. Lu, C. Y. Pei, S. X. Xu and S. L. Qiu, *Chem.–Eur. J.* **18**, 10250–10253 (2012).

[2.14] R. Y. Tsien, *Annu. Rev. Biochem.* **67**, 509–544 (1998).

[2.15] H. Morise, O. Shimomur, F. H. Johnson and J. Winant, *Biochemistry* **13**, 2656–2662 (1974).

[2.16] H. X. Deng, S. Grunder, K. E. Cordova, C. Valente, H. Furukawa, M. Hmadeh, F. Gandara, A. C. Whalley, Z. Liu, S. Asahina, H. Kazumori, M. O'Keeffe, O. Terasaki, J. F. Stoddart and O. M. Yaghi, *Science* **336**, 1018–1023 (2012).

[2.17] J. X. Liu, B. Lukose, O. Shekhah, H. K. Arslan, P. Weidler, H. Gliemann, S. Brase, S. Grosjean, A. Godt, X. L. Feng, K. Mullen, I. B. Magdau, T. Heine and C. Woll, *Sci. Rep.* **2**, 921 (2012).

[2.18] M. Kondo, S. Furukawa, K. Hirai and S. Kitagawa, *Angew. Chem., Int. Ed.* **49**, 5327–5330 (2010).

[2.19] D. Maspoch, D. Ruiz-Molina, K. Wurst, N. Domingo, M. Cavallini, F. Biscarini, J. Tejada, C. Rovira and J. Veciana, *Nat. Mater.* **2**, 190–195 (2003).

[2.20] C. Mellot-Draznieks, C. Serre, S. Surble, N. Audebrand and G. Ferey, *J. Am. Chem. Soc.* **127**, 16273–16278 (2005).

[2.21] C. Serre, C. Mellot-Draznieks, S. Surble, N. Audebrand, Y. Filinchuk and G. Ferey, *Science* **315**, 1828–1831 (2007).

[2.22] M. Kim, J. F. Cahill, H. Fei, K. A. Prather and S. M. Cohen, *J. Am. Chem. Soc.* **134**, 18082–18088 (2012).

[2.23] J. An and N. L. Rosi, *J. Am. Chem. Soc.* **132**, 5578–5579 (2010).

[2.24] B. L. Chen, S. Q. Ma, F. Zapata, F. R. Fronczek, E. B. Lobkovsky and H. C. Zhou, *Inorg. Chem.* **46**, 1233-1236 (2007).

[2.25] M. E. Zaballa, L. A. Abriata, A. Donaire and A. J. Vila, *Proc. Natl. Acad. Sci. USA*, **109**, 9254-9259 (2012).

CHAPTER 3

METAL–ORGANIC FRAMEWORKS CAPABLE OF HEALING AT LOW TEMPERATURES

3.1 Introduction

Metal-organic frameworks (MOFs) are crystalline materials whose large lattice parameters afford them organized pores spacious enough to trap small molecules of gases for energy storage[3.1-3.8] or oligomers for drug delivery[3.9-3.12]. In contrast to other types of porous materials like silicates[3.13-3.16] or crosslinked polymer networks[3.17-3.22] whose constituent components interact mainly through covalent bonds, slightly ionic nature of the building blocks in MOFs makes these solid frameworks vulnerable or dynamic when exposed in a polar liquid environment. To name a few, organic linkers in MOFs can be replaced by simply soaking the crystals in a fresh ionic solution[3.23-3.25]; the exchange of metal centers in frameworks allows the synthesis of alloy-like solids[3.25-3.27]; the unloading of organic linkers triggers a sustained robotic motion in liquid/air interface[3.28]; and the transformation of shiny crystals into hairy ones enables a sequential trapping of proteins[3.29]. Nevertheless, these interesting phenomena all suggest extreme dynamic or chemical instability of framework structures. Is there any way to reverse this process or heal the corroded crystals by patching the escaped building blocks back to the original framework structure? Since extensive research or designs in MOFs have been geared toward heterogeneous catalysis in chemical reactors, healing of these crystalline solids will undoubtedly extend their operation lifetime; when an inter-solid healing or binding is made possible, a dense packing could even promote the toughness of the solid, facilitating multiple cycles of loading and unloading.

Moreover, many efforts in these man-made crystals have been placed upon constructing unique bulk crystalline structures, while very little attention is paid on outer surfaces of the solids. Since MOFs share similar mechanical behaviors[3.30- 3.31] as rigid solids of ceramics, lack of rotational freedom in their stacking[3.32] or a slight variation in crystal growth could yield many small crystal grains atop the solid surfaces. When an uneven volume change or a crack is initiated, it could lead to catastrophic failure over a short period of time. Will the proposed healing effectively weld the small grains into larger ones? While we know eutectic solids can weld at a lower temperature than their individual components alone, phase transitions in many metal-containing solids or oxides occur at a temperature way beyond service conditions available[3.32]. If rather low interface mobility from the atomic or ionic constituents has hindered the interfacial regroupings, will a stronger mobility from MOFs effectively lower the resulting temperature in healing?

To shed light on aforementioned questions, we report our work in healing crystalline solids of MOFs, where tiny grains on bulk solid surfaces can merge into larger ones at a temperature of -56 °C (45 °C lower than that of eutectic solids of Na/K[3.33] and 80 °C lower than room temperature synthesis of one recent MOFs[3.34]). When our MOFs are crushed into powders, the resultant solid membranes show enhanced toughness or bendability after healing; when pristine crystals are healed, a 300 % leap in modulus and 250 % in hardness are observed.

3.2 Materials and Method

3.2.1 Materials

Cu-MOFs crystals: Cu-MOFs were synthesized using the “shake-and-bake” method described in section 2.2.1.

Cu-MOFs powders and membranes: The powders were prepared by mechanically crushing as-synthesized Cu-MOFs crystals using a mortar and pestle. The Cu-MOFs membrane (>15 μm in thickness) for nanoindentation tests was obtained by drop-casting a 2.0 wt% water suspension of Cu-MOFs powders onto a piece of Si wafer. The Cu-MOFs membrane (~ 25 μm in thickness) for three-point bending test, on the other hand, was prepared by drop-casting 5.0 wt% Cu-MOFs powder suspension on a Teflon film. After water was naturally dried, a freestanding membrane was harvested directly from the Teflon substrate.

Healing of Cu-MOFs membranes: The membrane for nanoindentation tests was obtained after dropping 0.2 mL of DEF (>99.0%, TCI America) over drop-casted MOF powders on Si wafer and let it dry in air. The healing of membrane (freestanding) was by soaking the freestanding piece in DEF for 12 hours.

Cu-MOFs/PDMS composite: Bubble-free fresh PDMS precursor was casted atop a polycarbonate film to form a thin layer with a thickness of 1 mm. Crushed Cu-MOFs powders were then laid on this uncured PDMS surface, letting the solid sink to the bottom of the viscous liquid. Final Cu-MOFs/PDMS composite was received after annealing the mixture in the convection oven at 100 ^{o}C for 1 hour.

3.2.2 Characterization and Modeling

Sample characterizations: X-ray diffractometry (XRD, Rigaku Multiflex) was conducted to examine the crystal structures of the as-synthesized and healed Cu-MOFs samples. The weighted average wavelength of the CuKα x-ray source was 1.5417 Å. Atomic force microscopy (AFM, Dimension 3100 SPM system) was performed to reveal the surface morphology of the Cu-MOFs and its inner layers under air and water environment. The Scanning Probe Image Processor software (SPIP, Image Metrology, Denmark) was used to analyze the depth profiles. Optical

images of the Cu-MOFs were taken with an optical microscope (Meiji ML8000) equipped with a digital camara (Moticam 2000). Proton Nuclear magnetic Resonance (^{1}H-NMR) spectra were obtained with a 500 MHz NMR spectrometer (Bruker AVANCE DRX) with deuterated water as the solvent. DSC experiments were conducted on a Thermal Analysis Instruments heat flow calorimeter (DSC 2920 CE), which was calibrated using indium as reference material by matching the melting temperature and the associated enthalpy of indium. Heat flow was recorded when 0.005 mL of DEF was added into 3.498 mg of Cu-MOFs and 0.01 mL of water in every 15 min. For the control test, the Cu-MOFs powder was removed, and 0.005 mL of DEF was added into 0.01 mL of water in every 15 min. All experiments were performed under a nitrogen atmosphere.

Micro- and macro-scale mechanical tests: Nanoindentation tests were performed using Hystron Bio-Ubi. Quasi-static ‘‘trapezoidal’’ load function tests were selected, with a 5-s loading, 2-s holding, and a final 5-s unloading. Maximum forces were set at 50 μN for samples made from crushed Cu-MOF powders and 100 μN for bulk crystals. The tip used is a Berkovich tip with a tip radius of 70 nm. 3 samples, with 3 zones in each sample and 3 well-separated points in each zone, were tested before and after healing, respectively. In three-point bending tests, samples were placed between two glass slides with a spacing of 3 mm. Then carefully weighted small Si pieces were gradually loaded on top of each sample to investigate the maximum deflection before a failure.

Calculation of domain size: The crystalline size of monoclinic phase was computed by Scherrer’s equation as: $D = 0.9\lambda/(\beta\cos\theta)$, where D is the grain size, λ is the radiation wavelength (1.5417 Å), β is the full width at half maximum (FWHM), and θ is the diffraction peak angle. The calculated grain sizes are based on the XRD data shown in Figure 3.2F and 3.6D.

Modeling methods: A Modified CVFF force field was employed in the molecular dynamics (MD) simulations with flexible SPC model for water. The systems were equilibrated in the canonical ensemble at 300 K. The simulation time was 1.0 ns with a time interval of 1.0 fs. All the simulations were performed with Discover module in Material Studio package.

3.3 Results and Discussion

3.3.1 Healing of Mechanically Crushed Cu-MOF Powders

We chose Cu-MOFs $(Cu(FMA)(4,4'\text{-}Bpe)_{0.5}\cdot 0.5H_2O)$[3.35] that was synthesized with an interpenetrating pseudo-cubic crystal structure (Figure 2.2 and Table 2.1) to demonstrate our healing concept. Before any healing process is allowed to occur, these MOFs are briefly dispersed or soaked in water. Once sufficient amount of building blocks are released into this polar media, water is removed by natural drying and diethylformamide (DEF, $(C_2H_5)_2NCOH$) is added as the healing agent. Figure 3.1A demonstrates the healing of a broken elastic membrane by applying crushed powders of these MOFs. When the elastic membrane was cut with a blade (Figure 3.1A-left), the open wound was briefly treated with DEF. Shortly after, the membrane recovered most of its bendability in ambient conditions (Figure 3.1A-right). Another freestanding leaf-like structure molded from MOF powders (Figure 3.1B) was even capable of withholding a high temperature up to 200 ^oC without a major breakdown. Both microscopic (nanoindentation) and macroscopic experiments (three-point bending) were utilized to evaluate the membrane before and after the DEF treatment (Figure 3.1C). As the nanoindentation load increased, the physically aggregated membrane behaved rather fragile (upper inset), where failure marks on the loading curve suggest easy and permanent deformations.

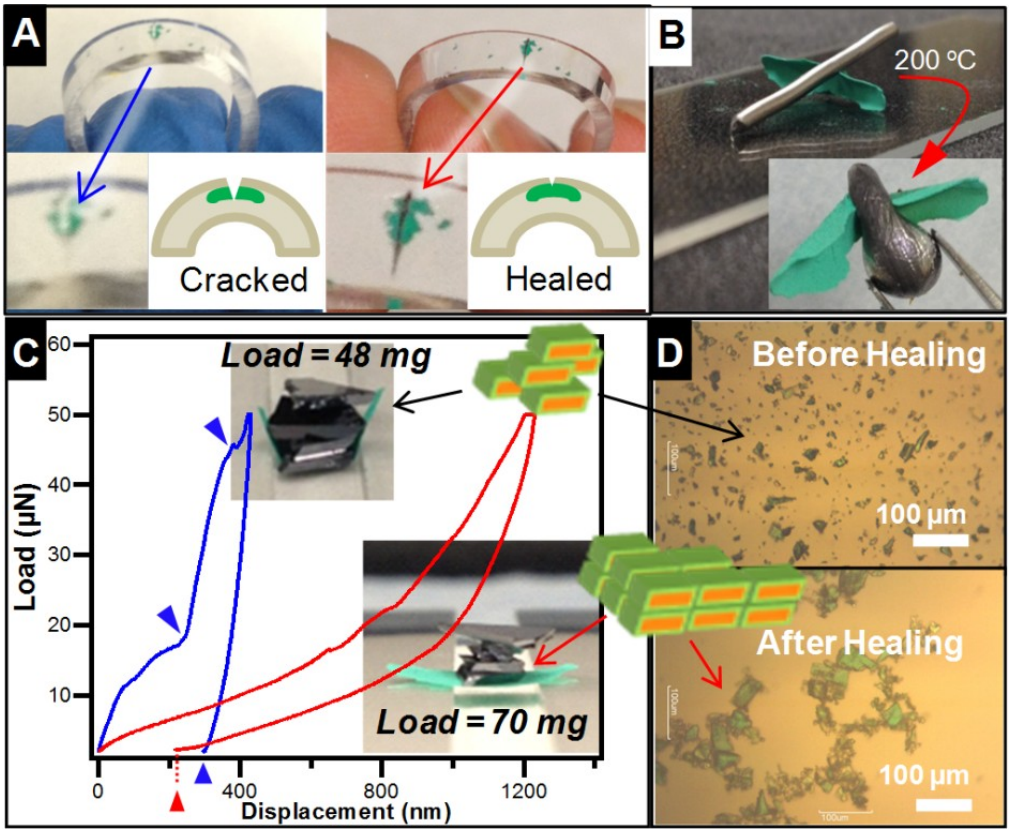


Figure 3.1 Mechanically crushed Cu-MOFs powders capable of healing at ambient conditions. (A) Snapshots and schematics of a MOF/PDMS composite showing diminished wound after DEF treatment; (B) MOF can be molded into a freestanding 3D object by keeping the shape even after annealing at 200 ^{o}C, at which a solder rod melts; (C) Typical load-displacement curves of the MOF membranes before (blue) and after (red) healing. Failures of the unbound membrane are marked by blue triangles. The picture insets record the maximum loadings (48 vs. 70 mg) that the membranes (unbound vs. bound) bear in a three-point bending test; and (D) Optical microscope images of well dispersed MOF crystals in water (top) and the highly networked ones after DEF addition (bottom).

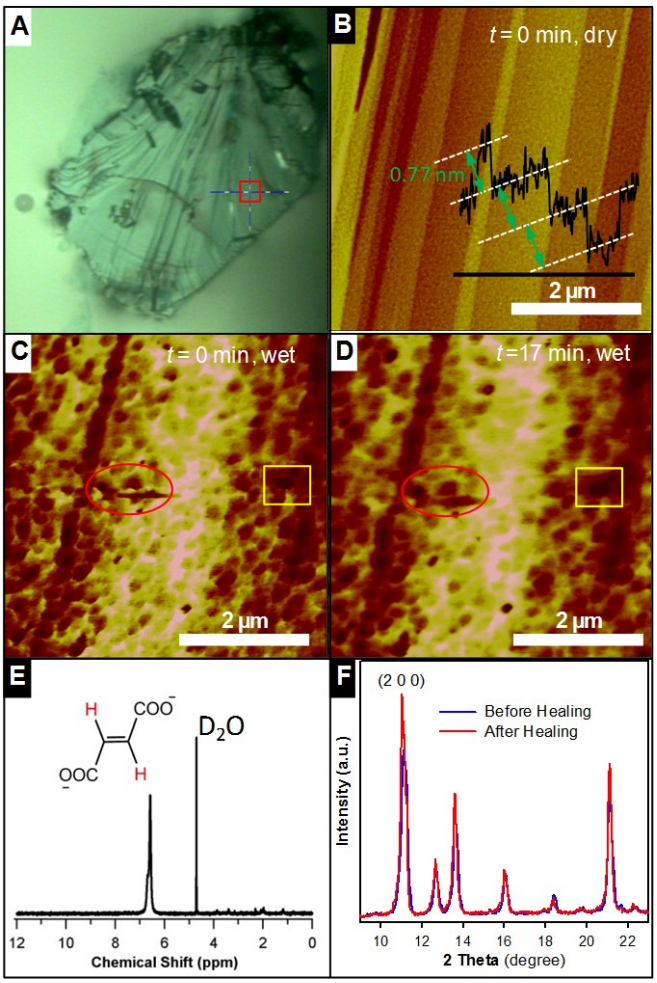


Figure 3.2 Surface morphology of one piece of crushed Cu-MOFs and its dynamic evolution in liquid. (A) Optical microscope image of the crushed crystal with dense surface terrains; (B) AFM image revealing the surface terraces with a fringe spacing of approximately 0.77 nm, matching the interplanar distance of (200) planes; (C-D) AFM images recording the morphological evolution of the water-soaked surface at 0 and 17 min; (E) ^{1}H-NMR spectrum of the clear solution filtered from a mixture of MOF powders and D_2O after a 24-hour impregnation; and (F) Powder XRD pattern of the MOF powders before and after DEF treatment.

In contrast, the healed membrane easily deflected with a smooth loading curve by retaining little residual strain. If we pay closer attention to the crushed pieces of these crystals, they were loosely dispersed in water with their morphologies remaining intact for extended

period of time (Figure 3.1D). Immediately upon the addition of DEF, dispersed solids bundled together by forming network-like aggregates, implying the anisotropic nature of Cu-MOFs during this healing process. We can gauge this anisotropy from the Cu-MOFs by looking at the compositions of their major crystal planes (Table 2.1). The frameworks are essentially organized via noncovalent bindings, either a copper-oxygen (Cu-O) or copper-nitrogen (Cu-N) bond, where the latter is much weaker than that of Cu-O[3.36]. As a consequence, the (200) planes have the lowest interlayer strength, indicating a low driving force to form the bonds or less stability. If we crush a bulk crystal of such into fine pieces, we would expect the outer surfaces of the powders to be mainly (200)s. Indeed, AFM scanning revealed layered structures with individual layer thickness of 0.77 nm, closely matching 0.80 nm of lattice spacing for (200)s (Figure 3.2A and B). The dynamic changes on this (200) were caught after soaking the sample in water, supported by *in situ* AFM imaging at different times (Figure 3.2C and D). Right after water covered the crystal surface, many small holes appeared; after 17 minutes, they became deeper and larger (marked by the red ovals and yellow rectangles in Figure 3.2D), indicating a continuous dissolution of substances from the surfaces into the aqueous medium. When the water-soluble substance was extracted, ^{1}H-NMR confirmed the existence of fumarate (s, d = 6.7 ppm) as shown in Figure 3.2E. Collectively, both the AFM and NMR studies have pointed to one fact that the water-soluble fumaric acid or fumarate moieties roll off from (200) surfaces, destabilizing the Cu-MOFs crystal in a continuous manner. Naturally, the following question comes with the role of the liquid (DEF) in the healing process. A brief comparison in diffraction patterns of crushed solids before and after DEF treatment suggests peak sharpening at (200)s only (Figure 3.2F, Table 2.1). Since the peak intensity or sharpness is indicative to the

perfectness of the inter-planar stacking, a sharper (200) peak unambiguously indicates a promoted crystal alignment and structure perfection along [200] direction (Table 2.1).

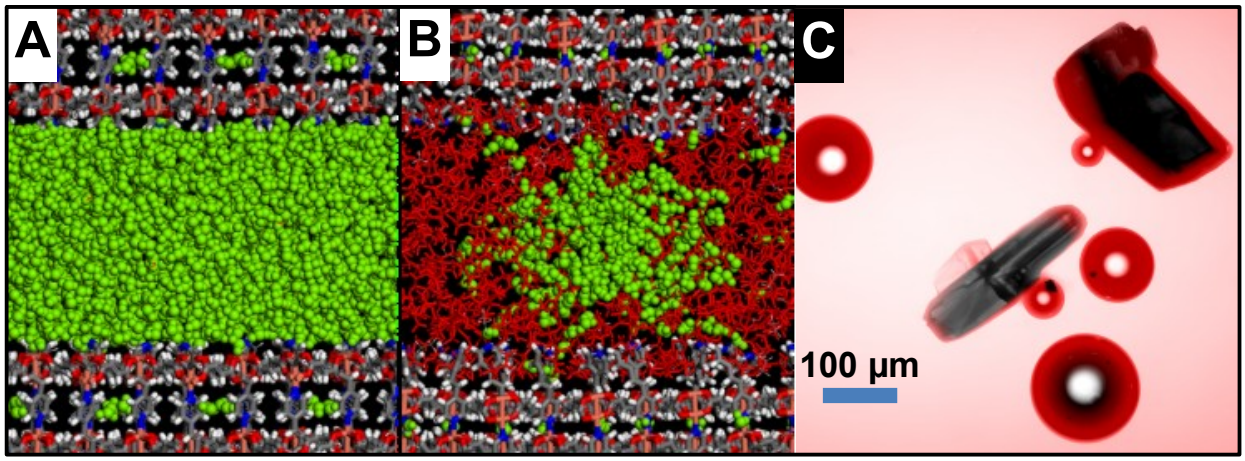


Figure 3.3 Molecular dynamics simulation of the interfacial healing process and optical image of bubbles generated during the DEF treatment. (A) Ball and stick model of water molecules on the (200) surface of Cu-MOFs; (B) Structure model of water dewetting on (200) after DEF is involved. Water molecules are represented as green balls and DEF as red sticks; and (C) laser scanning confocal microscope image capturing water bubbles adjacent to the bulk crystal of MOFs after DEF incorporation at room temperature. Fluorescent dye is used in the imaging process for visual convenience.

Even though DEF can patch fresh (200)s with building blocks (fumarate), an energy release is still required to ensure the tight connection between (200)s. Since the Cu-MOFs were synthesized from an aqueous environment, it is not surprising that the exposed (200)s are covered with a thin layer of water molecules (in green) (Figure 3.3A). A careful view suggests that the water molecules are largely contained between the top boundaries of the (200) surfaces only, creating a pseudo-stable configuration. As water is not a good solvent for dangling building blocks (4, 4'-Bpe), presence of water layer alone blocks the diffusion of these 4, 4'-Bpe and inhibits the binding of (200)s. When amphiphilic molecules (DEF in Figure 3.3B: red sticks) are poured into this system, the nonpolar branches (two ethyl groups) from the DEF are quickly

inserted in between the (200)s by depleting the surface waters and also distorting the surface frameworks to a certain degree (Figure 3.3B). Essentially, strong affinity of the surface components to bipolar molecules like DEF has destabilized the original surface configuration.

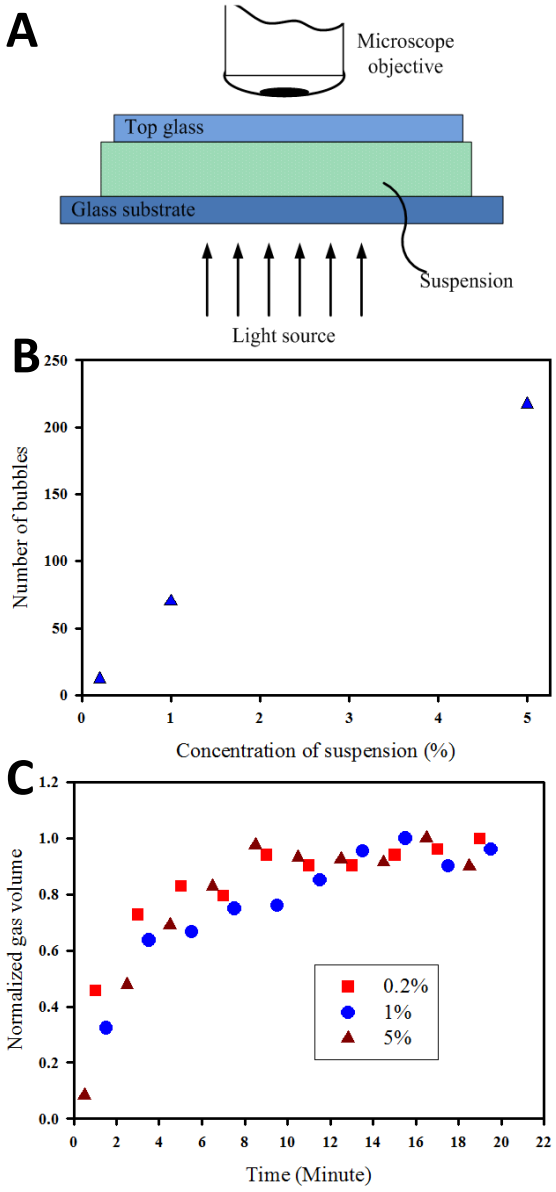


Figure 3.4 Dynamical process of water bubbling after adding DEF into an aqueous suspension of Cu-MOFs. (A) Schematic illustration of the experimental setup showing the suspension is sandwiched between a cover glass and a glass substrate; (B) The number of bubbles vs. concentration of suspension plot indicating a linear relationship; and (C) Normalized gas volume vs. time plots for the 0.2%, 1% and 5%wt MOF suspension.

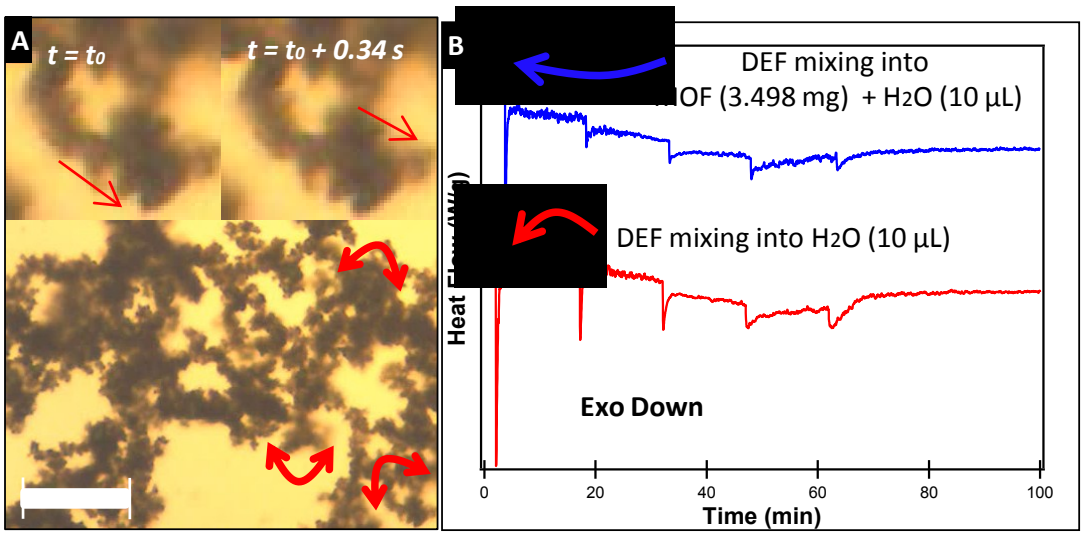


Figure 3.5 Networking of fine pieces of crushed Cu-MOFs. (A) Optical microscope images unveiling quick but sustained vibrations when dispersed MOFs were mixed with DEF. Scale bar: 100 µm; (B) DSC curves revealing heat release (initial step) after mixing DEF with MOFs/water. Sudden spikes were due to the dropping of the liquid into the DSC pan.

As a result, large energy release from this DEF engagement is likely to promote Cu-N restructuring, or the binding of Cu and the dangling 4, 4'-Bpe. As evidenced by our experiment shown in Figures 3.3C and 3.4, water vapors from the Cu-MOFs/water/DEF mixture were frequently caught, suggesting local heating a plausible mechanism for the observed binding. Another piece of evidence is revealed as water-dispersed solids were mixed with a few droplets of DEF, quick vibrations of the solids followed by an immediate networking were recorded (Figure 3.5-left panel). While liquid diffusion around the solids can introduce a certain level of dynamic fluctuation, the long lasting vibration is perhaps closely related to the local heat release (Figure 3.5-right panel). Essentially, this local heating has driven off the water molecules from the cleaved (200)s and mobilized the metal centers or organic linkers. To be noted, this DEF-assisted interfacial binding is different from the traditional solvent exchange or extraction/insertion processes[3.24]. As the DEF molecule (4.5 Å) is considerably larger than the

pore size of Cu-MOFs (3.6 Å)[3.35], it will not intrude the bulk frameworks. Correspondingly, we only observed a relative change in peak intensities on XRD, with no alteration in any of the peak positions.

3.3.2 Defect Repairing on Cu-MOF Bulk Crystals

While defects or dangling bonds can be created mechanically or by solvation in an aqueous environment, freshly synthesized MOF crystals are not perfect either. If crushed solids or powders can heal, this process might even repair defects on as-synthesized bulk crystals (Figure 3.6). We used nanoindentation as an effective tool to evaluate binding-induced healing in surface deformation. Unlike mechanical tests for crushed membranes in Figure 3.1C, neither of our samples here showed permanent deformation. Rather smooth curves were observed in both loading and unloading regions, where the modulus jumped from 4 to 12 GPa and hardness from 400 to 1000 MPa. Since the statistical plot in Figure 3.6B suggests this as a general trend, we tend to believe this healing or welding process once again is closely resulted from structure transition on surfaces. As revealed in Figure 3.6C, before the surface healing, the Cu-MOFs surface was composed of many small grains with an average size of 10 nm, after the healing the average diameter increased drastically to 30 nm. The XRD patterns (Figure 3.6D) before and after this transition suggest these merging occur mostly in (200) and (002) planes, as evidenced by the peak sharpening and also the increased domain sizes estimated from Scherrer's equation (Table 3.1). When crystal grain size increases, the surface area and the surface defect density decrease, rendering less scattering in measurement (Figure 3.6B, upper corner) and a much-enhanced mechanical performance.

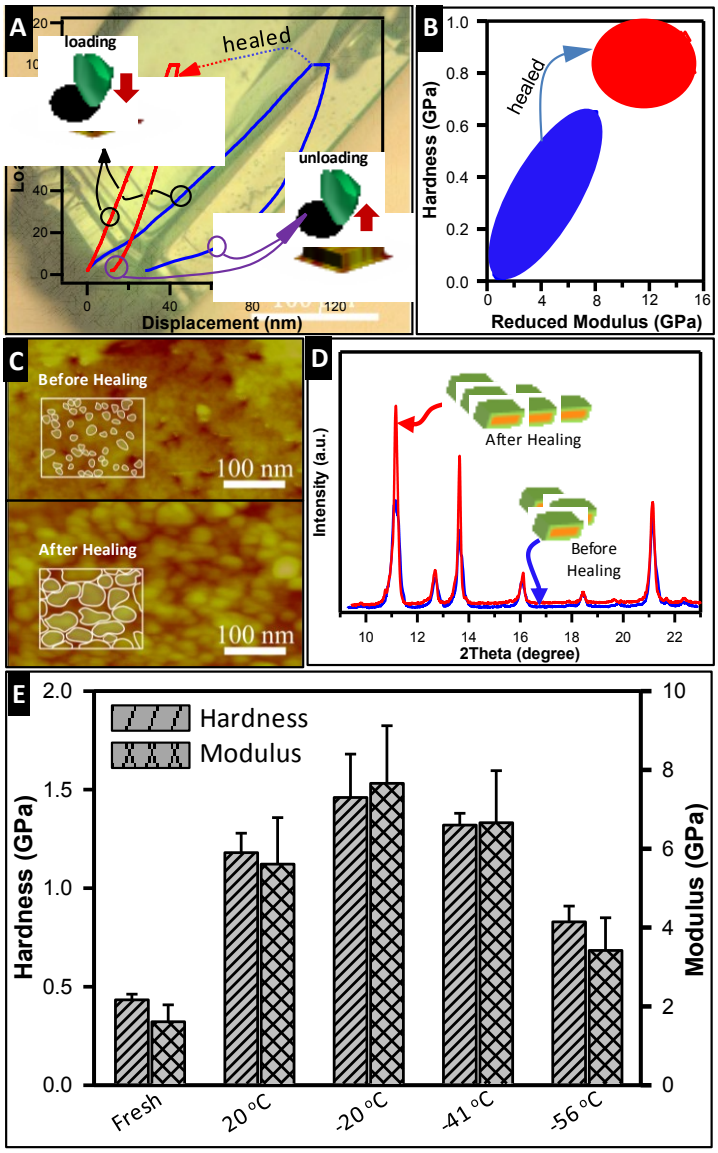


Figure 3.6 Bulk crystal of Cu-MOFs capable of surface healing. (A) Load-displacement curves of the fresh and bound MOF crystals; the optical image of one crystal is set as background; (B) Hardness-reduced modulus plot of the nanoindentation results of the fresh (blue) and repaired (red) crystal revealing statistical leap in deformation resistance; (C) Powder XRD patterns of the fresh and repaired crystals; and (D) AFM images of the crystal surfaces showing increased grain sizes after surface healing. (E) Modulus and hardness histograms extracted from groups of nanoindentation showing enhanced mechanical properties after healing at low temperatures. Note: Cold DEF at -20, -41, and -56 ^{o}C was obtained by placing glass vials containing DEF in a cold bath of sodium chloride/ice (w/w = 1/3), acetonitrile/dry ice, and octane/dry ice, respectively.

Table 3.1 Change in grain size during the healing

Samples	Peak	Grain size (nm)	
		Before healing	After healing
Cu-MOFs powders	**(200)**	**22.8**	**29.6**
	(002)	**33.4**	**34.8**
Cu-MOFs crystals	**(200)**	**23.9**	**46.3**
	(002)	**33.5**	**66.6**

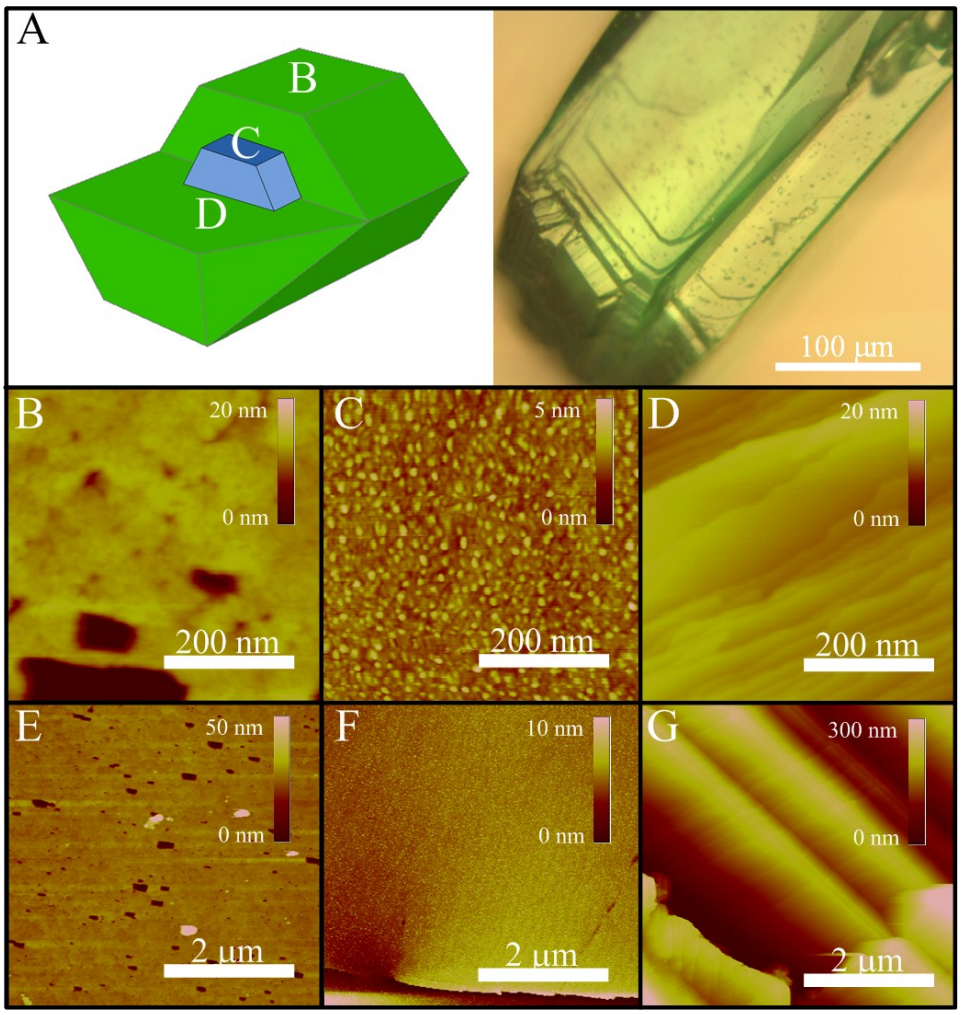


Figure 3.7 Cu-MOFs show a heterogeneous structure. (A) Schematic model (left) and an optical microscope image (right) of the MOF crystal. AFM images of outer surfaces (B&E) showing most defects, the center surfaces (D&G) with dense structures, and intermediate surfaces (C&F) with rather a smooth topography.

From the structure point of view, the surface healing or welding has tightened the dangling Cu-N bonds on the MOFs surfaces. As illustrated in Figure 3.7, MOF crystals bear a heterogeneous layered structure. We postulate <022> as the preferred orientation for bulk crystals, where each (022) of the unit cell contains 8 Cu-O and 4 Cu-N bonds. As evidenced by Figure 3.7B, abundant defects or etch pits are clearly visible all over the (022), giving us the opportunity to see repaired surfaces with much increased grain sizes (Figure 3.6C). However, if those tiny grains are merged through the Cu-N directions only, then longitudinal shaped or rod-like larger grains are expected. In contrast, what we saw in Figure 3.6C are rounded grains, suggesting the merging in more than one direction or healing of Cu-O bonds too. Indeed, the XRD patterns (Figure 4D) for DEF-treated bulk crystals show peak sharpening for both (200) and (002) planes. If we recall Cu-N and Cu-O as respective backbone along these two directions (Figure 2.2), this finding extends the dynamic binding from Cu-N to Cu-O even though the former is much easier to obtain and to engineer.

3.3.3 Surface Healing of Cu-MOFs at Low Temperature

To explore low temperature healing or welding, we soaked fresh crystals of Cu-MOFs in cold-baths of DEF at various low temperatures (-20, -41, and -56 ºC). We evaluated the surface healing on crystal alignment, grain size, and mechanical properties. Likewise, the diffraction peak of (200)s became sharper even after a cold-bath treatment at -56 ºC (Figure 3.8), where average grain size increased from 15 to 25 nm, accompanied by nearly doubled mechanical properties (Figure 3.6E). In comparison, soaking at a slight higher temperature of either -20 or -41 ^{o}C has tripled the hardness and modulus (Figure 3.6E). While the latter ones are not hugely different from those obtained in the ambient condition, substantial viscosity increase in DEF at -56 ºC has limited the diffusion of fumarate for reattachment of surface dangling bonds.

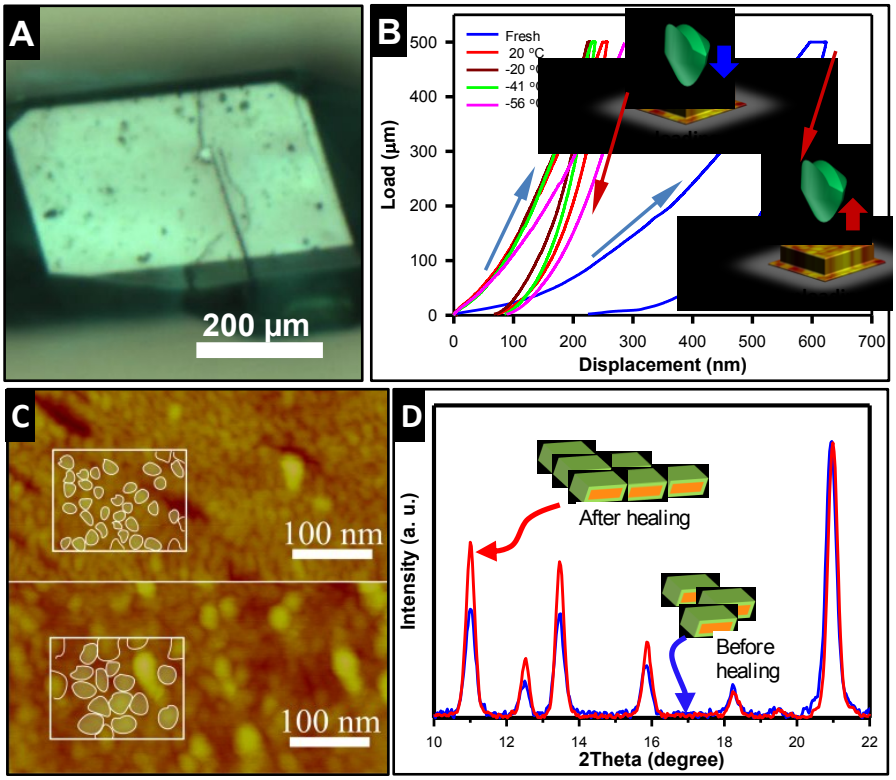


Figure 3.8 Surface healing of Cu-MOFs at low temperatures. (A) Optical microscope image of a crystal that was used for surface healing at low temperatures. This crystal was soaked in cold DEF (-56 ^{o}C) for 24 hours; (B) Load-displacement curves of the MOF crystals showing enhanced mechanical properties after healing at low temperatures; (C) AFM images of the fresh and cold-healed MOF (-56 ^{o}C) surface showing increased grain sizes; and (D) Powder XRD patterns of the unhealed and healed MOF crystals.

Nonetheless, these experiments suggest that low temperatures do not restrict the healing between crystalline gaps. It is worthwhile to note that there are already a few reports on room temperature synthesis of MOFs, using liquid phase epitaxy[3.37] or spray drying methods[3.34, 3.38]. While we cannot exclude the possibility of executing these ambient processes at even lower temperatures, the energy need to patch a crystal is certainly not at the same level as to assemble multiple types of building blocks for a 3D stacking. In addition, a lower temperature synthesis could occur at the expense of smaller crystals or much less surface areas. Furthermore, the

healing agent (DEF) that was used to patch imperfect crystals is not just a solvent; their amphiphilic nature has a tendency to remove surface water molecules and therefore, lowering the resistance for the acid linker (fumaric acid) to bond with the dangling base (4,4'-Bpe). As the latter process occurs with a thermal energy release, our patching is largely self-sustained. If those ambient syntheses follow a kinetic or thermal collision-dominant process, substantial resistance for a low temperature assembly is then expected.

3.4 Summary

In conclusion, crystals of metal-organic frameworks (MOFs) can be destabilized in a polar liquid environment by releasing building blocks that can be further patched back through a healing process at ambient and low temperature conditions. After healing, solid membranes exhibit greater mechanical resistance, which can potentially be molded into freestanding 3D objects or mend a broken elastic membrane. As patching crystalline gaps with small molecules largely drives these healing processes, sufficient mobility of the assisting reagents allows the patching to happen at a temperature as low as -56 ºC. If these healing phenomena are utilized for many other MOFs that are adopted as catalysts or hydrogen storage/separation materials, we expect their structure integrity can be kept after multiple cycles of packing or extensive uses. Furthermore, the knowledge gained will help us design future crystalline solids or ordered structures that can be assembled, repaired, or healed in various engineering applications.

References:

[3.1] L. J. Murray, M. Dinca, J. R. Long, *Chem. Soc. Rev.* **38**, 1294 (2009).

[3.2] M. P. Suh, H. J. Park, T. K. Prasad, D. W. Lim, *Chem. Rev.* **112**, 782 (2012).

[3.3] M. Dinca, J. R. Long, *Angew. Chem. Int. Ed.* **47**, 6766 (2008).

[3.4] N. L. Rosi, J. Eckert, M. Eddaoudi, D. T. Vodak, J. Kim, M. O'Keeffe, O. M. Yaghi, *Science* **300**, 1127 (2003).

[3.5] J. L. C. Rowsell, O. M. Yaghi, *Angew. Chem. Int. Ed.* **44**, 4670 (2005).

[3.6] B. Kesanli, Y. Cui, M. R. Smith, E. W. Bittner, B. C. Bockrath, W. B. Lin, *Angew. Chem. Int. Ed.* **44**, 72 (2005).

[3.7] T. A. Makal, J. R. Li, W. G. Lu, H. C. Zhou, *Chem. Soc. Rev.* **41**, 7761 (2012).

[3.8] J. Zhang, T. Wu, S. M. Chen, P. Y. Feng, X. H. Bu, *Angew. Chem. Int. Ed.* **48**, 3486 (2009).

[3.9] P. Horcajada, R. Gref, T. Baati, P. K. Allan, G. Maurin, P. Couvreur, G. Ferey, R. E. Morris, C. Serre, *Chem. Rev.* **112**, 1232 (2012).

[3.10] H. X. Deng, S. Grunder, K. E. Cordova, C. Valente, H. Furukawa, M. Hmadeh, F. Gandara, A. C. Whalley, Z. Liu, S. Asahina, H. Kazumori, M. O'Keeffe, O. Terasaki, J. F. Stoddart, O. M. Yaghi, *Science* **336**, 1018 (2012).

[3.11] P. Horcajada, C. Serre, M. Vallet-Regi, M. Sebban, F. Taulelle, G. Ferey, *Angew. Chem. Int. Ed.* **45**, 5974 (2006).

[3.12] P. Horcajada, C. Serre, G. Maurin, N. A. Ramsahye, F. Balas, M. Vallet-Regi, M. Sebban, F. Taulelle, G. Ferey, *J. Am. Chem. Soc.* **130**, 6774 (2008).

[3.13] H. Yang, N. Coombs, G. A. Ozin, *Nature* **386**, 692 (1997).

[3.14] D. Y. Zhao, J. L. Feng, Q. S. Huo, N. Melosh, G. H. Fredrickson, B. F. Chmelka, G. D. Stucky, *Science* **279**, 548 (1998).

[3.15] A. Stein, B. J. Melde, R. C. Schroden, *Adv. Mater.* **12**, 1403 (2000).

[3.16] Q. S. Huo, D. I. Margolese, U. Ciesla, P. Y. Feng, T. E. Gier, P. Sieger, R. Leon, P. M. Petroff, F. Schuth, G. D. Stucky, *Nature* **368**, 317 (1994).

[3.17] J. X. Jiang, F. Su, A. Trewin, C. D. Wood, N. L. Campbell, H. Niu, C. Dickinson, A. Y. Ganin, M. J. Rosseinsky, Y. Z. Khimyak, A. I. Cooper, *Angew. Chem. Int. Ed.* **46**, 8574 (2007).

[3.18] J. Schmidt, M. Werner, A. Thomas, *Macromolecules* **42**, 4426 (2009).

[3.19] R. Dawson, A. Laybourn, R. Clowes, Y. Z. Khimyak, D. J. Adams, A. I. Cooper, *Macromolecules* **42**, 8809 (2009).

[3.20] X. F. Ji, Y. Yao, J. Y. Li, X. Z. Yan, F. H. Huang, *J. Am. Chem. Soc.* **135**, 74 (2013).

[3.21] D. Q. Yuan, W. G. Lu, D. Zhao, H. C. Zhou, *Adv. Mater.* **23**, 3723 (2011).

[3.22] W. G. Lu, J. P. Sculley, D. Q. Yuan, R. Krishna, Z. W. Wei, H. C. Zhou, *Angew. Chem. Int. Ed.* **51**, 7480 (2012).

[3.23] H. H. Fei, C. H. Pham, S. R. J. Oliver, *J. Am. Chem. Soc.* **134**, 10729 (2012).

[3.24] M. Kondo, S. Furukawa, K. Hirai, S. Kitagawa, *Angew. Chem. Int. Ed.* **49**, 5327 (2010).

[3.25] S. M. Cohen, *Chem. Rev.* **112**, 970 (2012).

[3.26] X. Song, S. Jeong, D. Kim, M. S. Lah, *Crystengcomm* **14**, 5753 (2012).

[3.27] S. Das, H. Kim, K. Kim, *J. Am. Chem. Soc.* **131**, 3814 (2009).

[3.28] Y. Ikezoe, G. Washino, T. Uemura, S. Kitagawa, H. Matsui, *Nature Materials* **11**, 1081 (2012).

[3.29] G. H. Wang, Z. P. Xu, Z. G. Chen, W. Niu, Y. Zhou, J. T. Guo, L. Tan, *Chem. Commun.* **49**, 6641 (2013).

[3.30] U. Mueller, M. Schubert, F. Teich, H. Puetter, K. Schierle-Arndt, J. Pastre, *J. Mater. Chem.* **16**, 626 (2006).

[3.31] A. Kuc, A. Enyashin, G. Seifert, *J. Phys. Chem. B* **111**, 8179 (2007).

[3.32] W. D. Callister, *Materials Science and Engineering: An Introduction*, Wiley, New York 2007.

[3.33] F. Mannone, *Safety in Tritium Handling Technology*, Kluwer Academic Publishers, Luxembourg 1993.

[3.34] A. Carne-Sanchez, I. Imaz, M. Cano-Sarabia, D. Maspoch, *Nat. Chem.* **5**, 203 (2013).

[3.35] B. L. Chen, S. Q. Ma, F. Zapata, F. R. Fronczek, E. B. Lobkovsky, H. C. Zhou, *Inorg. Chem.* **46**, 1233 (2007).

[3.36] O. M. Yaghi, M. O'Keeffe, N. W. Ockwig, H. K. Chae, M. Eddaoudi, J. Kim, *Nature* **423**, 705 (2003).

[3.37] S. Bundschuh, O. Kraft, H. K. Arslan, H. Gliemann, P. G. Weidler, C. Woll, *Appl. Phys. Lett.* **101**, 101910 (2012).

CHAPTER 4

POROUS MATERIALS FOR ENERGY DELAY OF MILD SHOCK WAVE

4.1 Introduction

Many porous materials, including metal foams[4.1], honeycombs[4.2,] lattice solids[4.3], composite materials[4.4-4.6], have been used as protection medium against external load because of the capability of large volume decrease at constant pressure[4.1, 4.7]. Those materials usually have low relative density in the 0.1 – 10% range and densification strains could exceed 80%, which is capable of deforming greatly with a pressure of incoming shock wave above their yielding stress. In these energy absorption materials (EAMs), kinetic energy is dissipated by plastic deformation or stored elastically. On the one hand, the energy absorption pressure (P_w) must be as high as possible in order to reach a high energy absorption capacity (E) while lower than the impact load (P). On the other hand, P_w must be equal and lower than the pressure of transmitted wave (P_t) and as low as possible for safety.[4.8] The conflict of P_w makes it extremely hard to find a satisfactory solution if the required P_t is low and P is high. For example, in order to avoid traumatic brain injury (TBI)[4.9,4.10], the blast mitigation helmet must be able to reduce the high blast pressure (~ 10^2 psi) to a few psi using a thin, lightweight protection.

It is relatively well-known that shock waves propagating in porous materials are attenuated[4.11-4.12]. In shock wave application in physical therapy, the focused underwater shock waves interaction with human tissues is considered to cause tissue damage which is attributed to pressure amplification. Grinten[4.13] investigated the shockwave interaction with dry, water-saturated and partially saturated porous media, and found that the wave propagation through the rigid porous material is fully dispersed. Some others studied the thermodynamic processes of shock waves in bubbly liquid, dusty, and mist flow.[4.14, 4.15] In those cases, the entrapment of air

bubbles seem to significantly delay the shock wave pulse and also possibly increase the energy dissipation in the porous media.

Specifically, we are interested in searching for porous materials for mild shock (typically 0.1 ~ 1 MPa) delay or mitigation. Mild shocks exist in many forms, including striking our legs through body weight during a half Marathon race (~1 MPa over a few seconds), body-to-body collision in Husker football games (~0.2 MPa over a few tenth of seconds), air blasts in a battlefield (1 MPa over a few microseconds), supersonic release (0.1 MPa over a few microseconds) of projectiles from naval electromagnetic railguns, and many others. Mismanagement of these shocks can be annoying and severe, which leads to chronic pain in our body joints, lose memory/temper due to traumatic brain in juries, and debonding sensors inside projectiles.[4.20, 4.21] A good management of shock calls for reducing peak pressure in a timely fashion. Therefore, we are motivated to fabricate carbon aerogel with extremely high porosity to serve as the intervening medium to delay the shock wave pulse, thus greatly reduce the possible damage to the protected structure. Besides, a structured porous material is also made by absorbing graphene on glass microfiber filter, which is much more stable than aerogel under the shock wave. In order to investigate the effect of the pore size on shockwave delay, porous copper thin film with micrometer pores was also fabricated and characterized. Our porous copper is much more rigid and stable than the carbon aerogel with ~100 nm pores, which is promising for long-term use on shockwave delay.

4.2 Materials and method

Synthesis of giant graphene oxide (GGO): Graphite flakes (Sigma-Aldrich, cat #332461, ~150 μm flakes) were oxidized using the improved method.[4.16] A mixture of 72 mL concentrated H_2SO_4 and 8 mL H_3PO_4 (85 wt%) was added to a mixture of 0.6 g graphite flakes and 3.6g

$KMnO_4$. The reaction was stirred for 10 min at room temperature to obtain a greenish mixture. The greenish mixture was poured into 80 mL ice water with 1.0 mL 30% H_2O_2 and stirred for 2 hrs. The mixture was centrifuged (3300 rpm for 40 min), and the supernatant was decanted away. The remaining solid material was then washed in succession with 84 mL DI H_2O, 10 mL 30% HCl, 84 mL DI H_2O (multiple times till PH ~ 6) to obtain gel-like liquid. The gel-like liquid was washed with 20 mL ethanol (2×) then vacuum-dried at room temperature for 5 hrs, obtaining 0.48 g of product.

Synthesis of MWNT-COOH: MWNT-COOH (< 8 nm diameter, 10-30 um length) was purchased from Nanostructured & Amorphous Materials Inc. (CAS #99685-96-8). However, the MWNT-COOH was found to be hard to disperse in DI H_2O and was further treated using the method in Ref 4.17. 0.5853 g as-purchased MWNT-COOH, 5 mL HNO_3 (70 wt%), 15 mL concentrated H_2SO_4 (98 wt%) were mixed and sonicated for 10 min (40 kHz). The mixture was then stirred for 100 min under reflux. After cooling to room temperature, the reaction mixture was diluted with 25 mL of DI water and then vacuum-filtered through a filter paper (Whatman). The solid on the filter was slowly washed by DI water and filtered twice. The solid was collected from filter and was dispersed in 60 mL DI water. The suspension was centrifuged at 4000 rpm for 2 hrs and the top brownish supernatant was decanted away. The dispersion and centrifuge process was repeated on the black solid collected from the bottom. No solid was found to exist after the centrifuge and the black liquid was collected (PH ~ 5). The concentration of MWNT-COOH suspension was found to be 1.5 mg/mL by freeze-drying 2 mL suspension and weighing the left-over solid.

Preparation of ultra flyweight aerogels (UFAs): UFAs were prepared based on a "sol-cryo" method.[4.18] Typically, 0.5 mL 5 mg/mL GGO aqueous dispersion and 0.5 mL 0.26 mg/mL CNT

aqueous dispersion were added to a 3 mL glass vial. The mixture was stirred with a magnetic bar for 2 hrs, and then poured into the desired mold followed by freeze-drying for 16 hrs. The mixture was frozen in an ethanol bath immersed in liquid nitrogen bath. The as-prepared GGO/CNTs foam was chemically reduced by hydrazine vapor at 60 °C for 24 hrs, and finally 2 mg UFA (ρ = 2 mg/mL, f = 0.95). The density was calculated by the weight of UFA divided by the volume of initial liquid mixture assuming no shrink occurs during freeze-drying; f is defined as the weight ratio of GGO to CNT in preparation of the UFA. In our experiment, UFAs with different ρ and f were obtained.

Preparation of graphene/CNT absorbed filter: Glass microfiber filter (CAT No. 1823-035, d = 35 mm) and polypropylene filter (CAT No. 6788-1304, d = 13mm) were purchased from Whatman. Those filters were exfoliated to multiple layers and soaked into a suspension of 0.5 mg/mL GGO and 0.1 mL CNT for 24 hrs. The GGO/CNT absorbed filter layers were then freeze-dried for 16 hrs, followed by chemical reduction by hydrazine vapor at 60 °C for 24 hrs. The color of the filter papers turned from white to gray/dark during this process.

Preparation of porous copper thin films: Porous copper thin films were fabricated by colloidal templating followed by copper electrodeposition and template removal (Figure 4.1). Polystyrene (PS) microspheres (3 um) were purchased from Polyscience. 1%(w/w) PS microsphere suspension was drop-casted on a Au-coated Si wafer and dried in air to form colloidal template. The colloidal template was sintered at 95 °C for 2 hrs to enlarge the neck connecting the spheres. A uniform thin Au coating was chemically vapor deposited on Si wafer for 2 times 30 secs each. A conventional copper electrodeposition solution ($CuSO_4 \cdot 5H_2O$ 225 g L^{-1}, H_2SO_4 70 g L^{-1}) was used to deposit copper into the voids between spheres under constant voltage (1.0 V) using the simple two electronode setup. The PS colloidal template was removed by tetrahydrofuran,

leaving a copper porous structure. Finally, the porous copper thin film is immersed in an ethanol solution of n-tetradecanoic acid at room temperature for 3-5 days to change the pore surface from hydrophilic to hydrophobic.

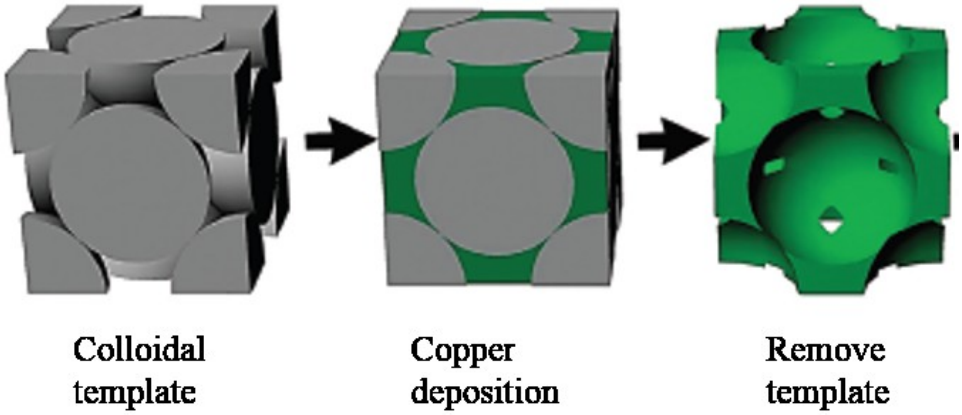


Figure 4.1 Sketch of the fabrication process of porous copper. First, colloidal template was fabricated by drop-casting 1 wt% polystyrene (PS) microsphere suspension on Au-coated Si wafer, followed by sintering at 95 °C for 2 hrs. Second, electrodeposition was then carried to fill the voids between the spheres. Last, PS colloidal template was removed by tetrahydrofuran, leaving a copper network structure. (Courtesy of Ref 4.22)

Characterization: Prepared samples were examined by FE SEM (FEI Quanta 200FEG).

Kolsky bar compression test[4.23]: The apparatus consists of two bars with a sample placed in contact between the bars. A stress wave, created on the incident bar, propagates to the sample where the wave splits into a reflected and transmitted wave. The stress in the sample is proportional to the transmitted wave.[4.19] In this study, we utilize a polymer cup filled with water and sample piece, which is placed between two bars. The tango black cup (Figure 4.2E) has the inner diameter of 18 mm, outer diameter 25 mm, and depth 4 mm, which can be fit into the aluminum sleeve (Figure 4.2D&E) perfectly.

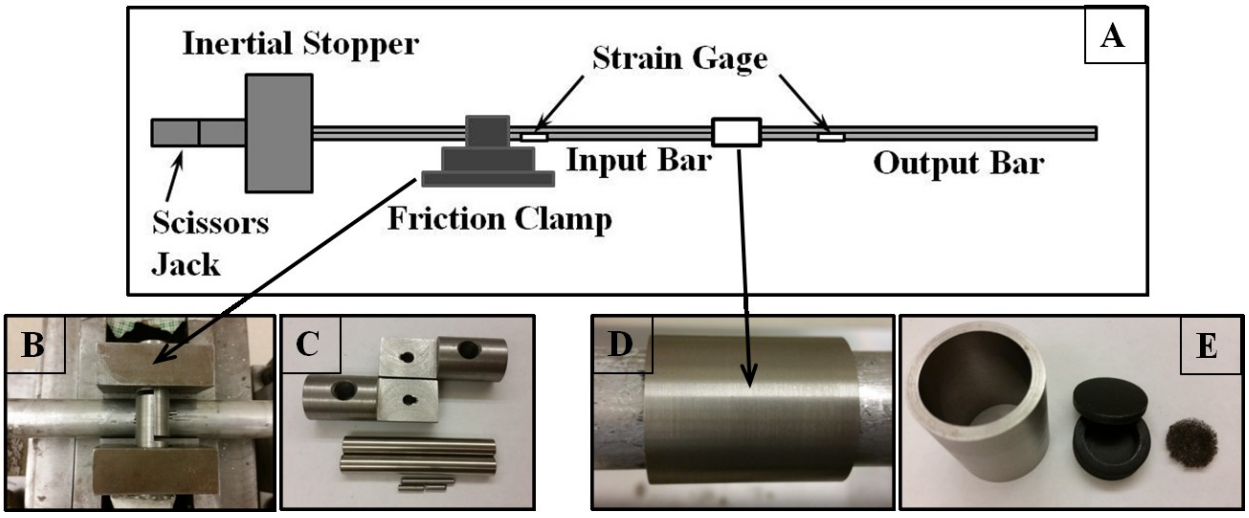


Figure 4.2 Sketch of the Kolsky bar compression test. The incident bar is clamped with a 130 lb mass onto one end (inertial stopper in panel A). A friction clamp is placed at a position that will give the desired pulse duration. By engaging the clamp and tightening a scissors jack between the mass and a loading support, the clamp-mass section of the incident bar is pre-compressed to the desired level. Forcing a pair of shear bolts that lock the clamp to a sudden break with a hydraulic system releases the pre-stored compression rapidly and thus generates a near square pressure pulse (Panel B and C). The large lock pins lock the shear bolts to the blocks at two ends and small breaking pins hold the shear bolts together at the middle before breaking. Tiny pieces of breaking pins are used as spacer to fix the breaking pin in position. The shock wave propagates through the incident bar, pressurizes the fluid and material within the sample cup impulsively (Panel D and E). The cup pressurization in turn initiates a pressure pulse propagating in the transmitted bar downstream. The strains associated with the pressure pulses in the bars are measured with compound high resistance strain gauges excited to 45 volts. The gauge signals are recorded with a digital oscilloscope at 1 MHz frequency as a loading history. The aluminum sleeve is made to fix the sample cup between two aluminum bars (Panel D). The tango black sample cup (panel E, made from 3D printing) has the inner diameter of 18 mm, outer diameter 25 mm, and depth 4 mm, which can be fit into the aluminum sleeve perfectly. Porous materials are trimmed to fit into the sample cup. The cup is filled with water and then sealed with glue before loading into the aluminum sleeve.

The two aluminum alloy bars (each 6-m long) are suspended by aligned brass bearings, and a cup containing the test sample piece is sandwiched between the bars (Figure 4.2A). The upstream bar is the incident bar with a 130 lb mass clamped onto one end. A friction clamp is placed at a position that will give the desired pulse duration. By engaging the clamp and tightening a scissors jack between the mass and a loading support, the clamp-mass section of the incident bar is pre-compressed to the desired level. Forcing a pair of shear bolts that lock the clamp to a sudden break with a hydraulic system releases the pre-stored compression rapidly and thus generates a near square pressure pulse (Figure 4.2B&C). The shock wave propagates through the incident bar, pressurizes the fluid and material within the cup impulsively. The cup pressurization in turn initiates a pressure pulse propagating in the transmitted bar downstream. The strains associated with the pressure pulses in the bars are measured with compound high resistance strain gauges excited to 45 volts. The gauge signals are recorded with a digital oscilloscope at 1 MHz frequency as a loading history.

4.3 Results and discussion

4.3.1 Fly-weight Graphene/Carbon Nanotube (GCNT) Aerogel

Figure 4.3 shows the digital images of prepared UFA (ρ = 2 mg/mL, f = 0.95). The UFA is only slightly heavier than the air at ambient conditions (1.2 mg/cm^3). The UFA also exhibits extremely strong elasticity, which can recover from the compression up to 70% of the original volume.

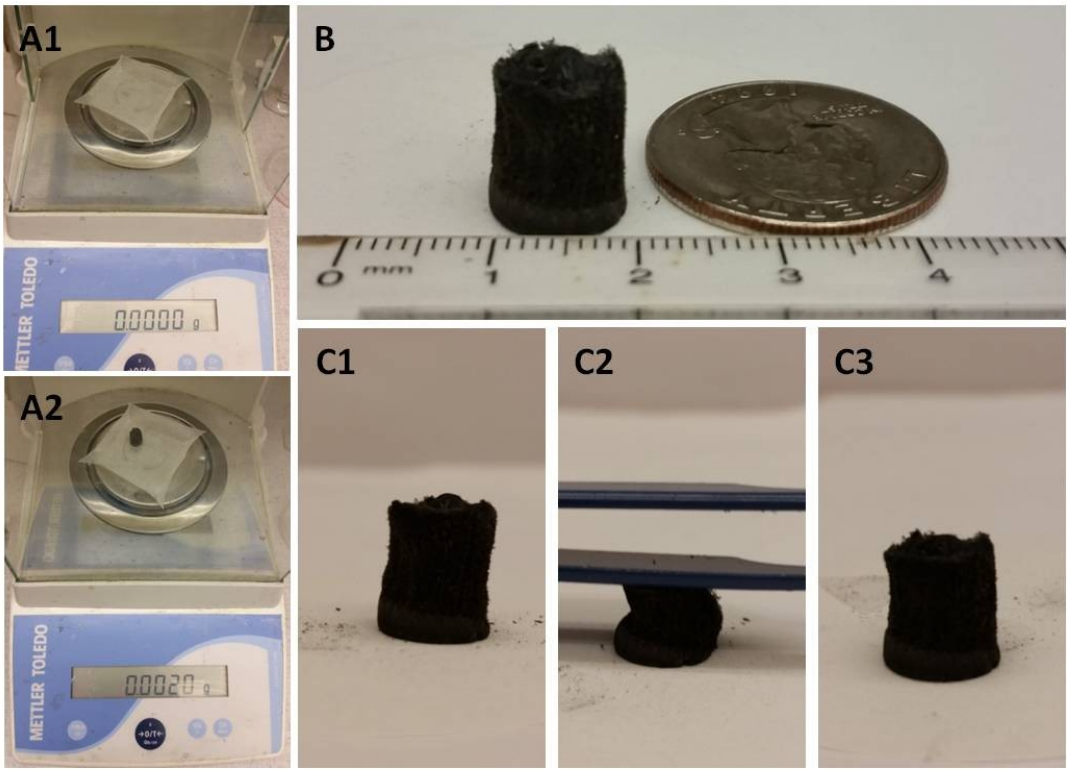


Figure 4.3 Digital images of prepared UFA (ρ = 2 mg/mL, f = 0.95). A) The weighting balance shows only 2 mg. B) UFA cylinder has diameter ~ 9mm and height ~ 16 mm. C) The compression and release process of UFA. The UFA is fully recovered (C3) to its original shape (C1) from a high deformation (C2).

Figure 4.4 illustrates the SEM images of the UFA. The porous architecture is formed through the interconnection between randomly oriented crinkly graphene sheets, with the pore size ranging from hundreds of nanometers to tens of micrometers. Zooming in reveals that those thin graphene sheets are connected mostly through twisting (Figure 4.4B). Possible CNTs were also observed to lie on the graphene sheets (Figure 4.4D). Direct deposition of graphene oxide onto aluminum foil reveals only tightly stacked graphene oxide sheets (Figure 4.5). Therefore, freeze-drying is a very effective method to fabricate graphene/CNT aerogel with extremely high porosity.

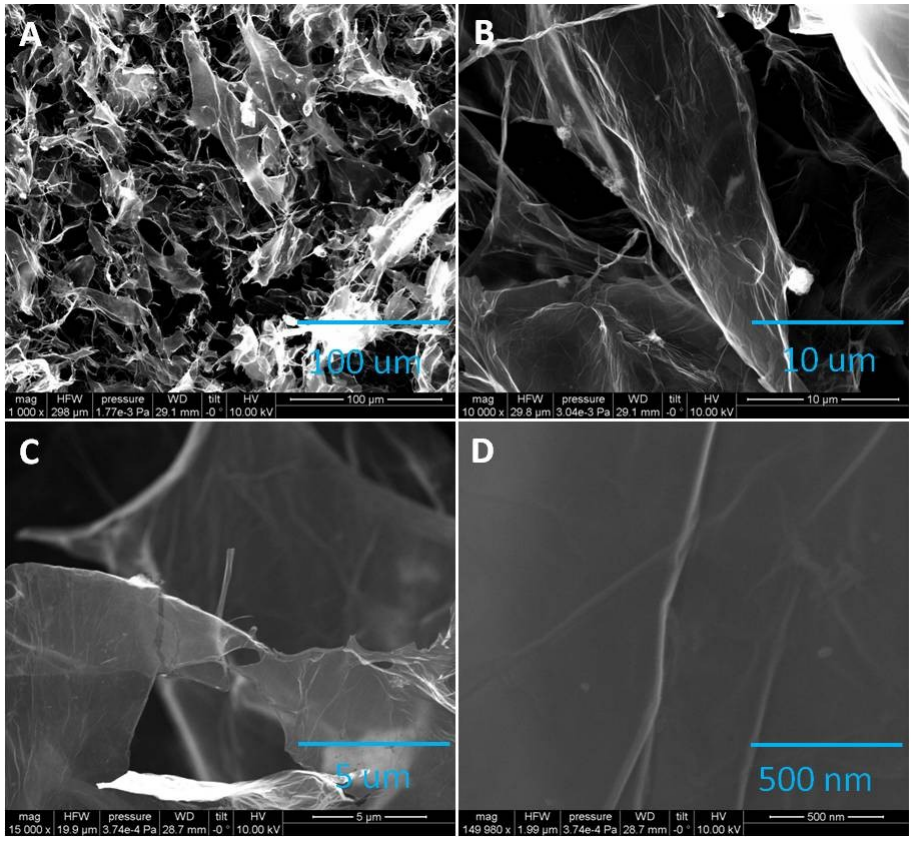


Figure 4.4 SEM images of the UFA. A) Porous architecture resulted from freeze-drying process. B) The GO sheets are tangled with each other. C) Part of the GO sheet was tear off and a 5 um hole was observed. D) Possible evidence of CNT lying on the GO sheet.

Oil absorption test also shows that the UFA can absorb a large amount of mineral oil and octane. While UFA is hydrophobic and cannot absorb water, UFA can still be easily tore apart when in contact with the water surface, indicating that the some locations of UFA still of hydrophilic, possibly resulted from partial reduction of graphene oxide.

Since our UFA is fragile, we compressed it into thin film (~ 3 mm) and stick it onto a layer of sticky tape. The UFA together with the tape was then gently loaded into the plastic cup using a soft piston. The cup was sealed with glue after filling up with water to preventing the induction of air bubbles into the water.

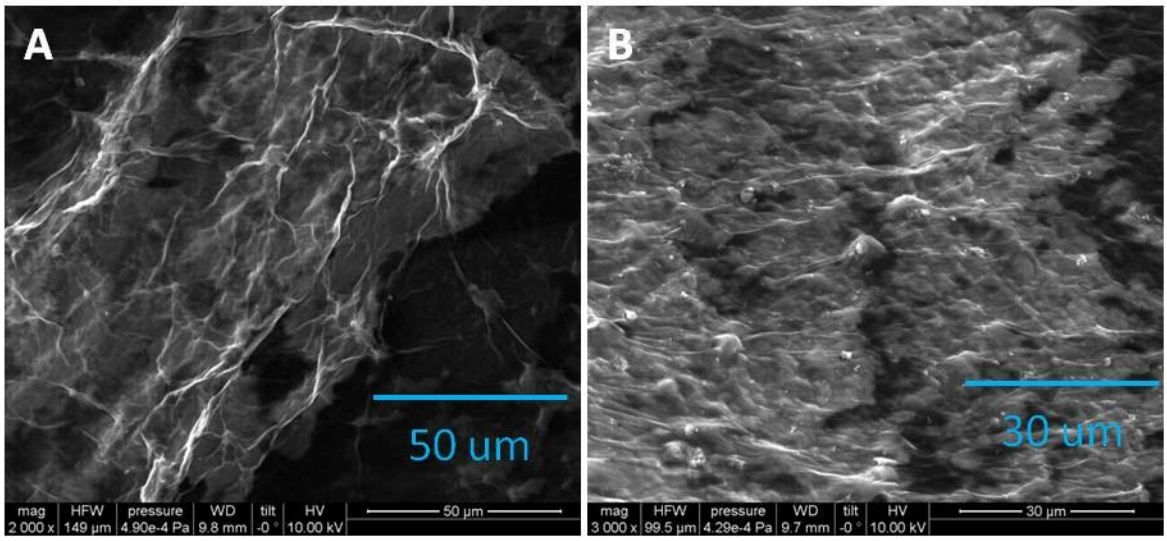


Figure 4.5 SEM images of graphene oxide deposited on aluminum foil. Graphene oxide sticks tightly to the aluminum surface and no porous structure was found.

Figure 4.6 illustrates the propagation of shock wave through the cup in the presence of UFA. A reference test was carried with cup filled with water only (red line – incident wave, green line – transmitted wave). In both cases, the incident waves are identical, which is a near square pulse with a width of 0.6 ms. When the cup is filled with water only, the transmitted wave is almost identical to the incident wave (green & red line). It takes only about 0.2 ms for the cup pressure to climb up to the incident pressure. This relatively small energy delay is most likely resulted from the response of the cup. On the other hand, when the cup is loaded with UFA, the cup pressure increases much slower as the shock wave pulse transmits through the UFA. In both cases, the transmitted wave has a significant drop at 0.6 ms due to the limited width of incident shock pulse. Ideally, if the incident square wave is wide enough, the pressure will continue to gradually increase to the maximum value and the exact delay can be measured.

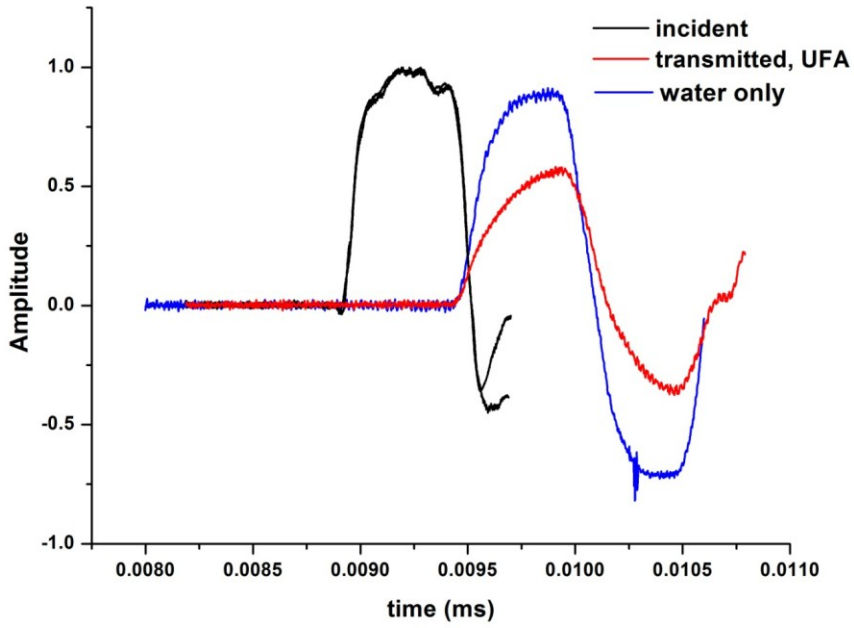


Figure 4.6 Effect of UFA on the transmitted wave through the cup. UFA was compressed into a thin layer before insert into the cup. The incident wave is a near-square pulse with a width of 0.6 ms. When the cup is filled with water only, the transmitted wave is almost identical to the incident wave (green & red line). It takes about 0.2 ms for the cup pressure to climb up to the incident pressure. This tiny energy delay is most likely resulted from some small air bubbles in water. On the other hand, when the compressed UFA was inserted into the cup and filled up with water, the cup pressure increases much slower while the shock wave pulse transmits through the UFA. In both cases, the transmitted wave has a significant drop at 0.6 ms due to the limited width of incident shock pulse.

When the thin film compressed from the UFA is loaded into the cup filled with water, a layer of air bubbles are trapped in the thin film due to its high porosity and hydrophobicity. The air bubbles reside on the hydrophobic porous surface and act as “hydrophilic” to the surface. The propagation of stress wave through air layer is much weaker compared to the propagation in liquid and solid. Therefore, significant delay can be observed in the porous hydrophobic thin film.

4.3.2 Graphene/Carbon Nanotube Absorbed Filter Paper

Although significant energy delay can be observed in the porous hydrophobic thin film, the thin film itself is still too fragile to serve as a proper media for energy delay of stress wave. Even the thin film seems to be smashed and wet after the Kolsky bar compression test. Therefore, a more robust material that can survive the shock wave must be considered. Inspired from the high durability of ordinary porous filter papers under the shock wave, we built the structured material with similar porosity by simply absorb the graphene/CNT onto the hydrophilic filter paper.

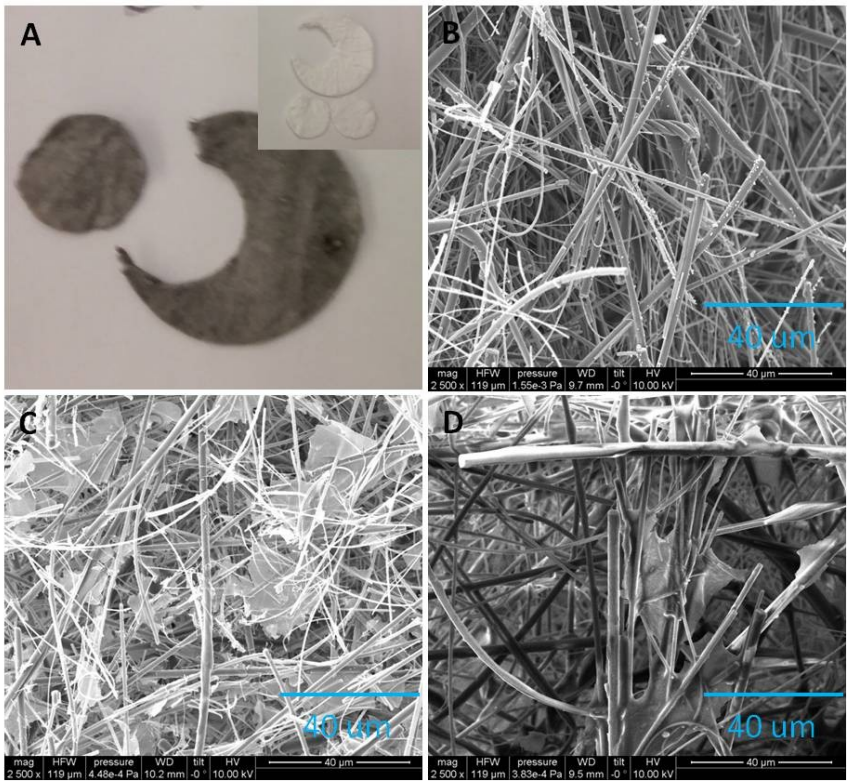


Figure 4.7 Graphene/CNT absorbed glass microfiber filter. A) Digital image, black color indicates the presence of graphene/CNT. Inlet – original white filter. B) SEM image of original filter. C) SEM image of graphene/CNT absorbed filter. The surface is covered with a layer of graphene sheets. Graphene sheets are embedded between fibers. D) SEM image of the filter after shock wave. Much less graphene sheets were observed. The graphene sheets look similar to those deposited on aluminum foil (Figure 4.4) and thicker.

Figure 4.7 shows the digital images as well as SEM images of the glass microfiber filter at different stages. After absorption, the original white filter paper became black, indicating the existence of graphene/CNT (Figure 4.6 A). The presence of graphene/CNT is verified by the SEM images of graphene/CNT absorbed filter paper. Very thin graphene sheets are found to be embedded between fibers. However, few graphene sheets were found on the filter paper after shock wave.

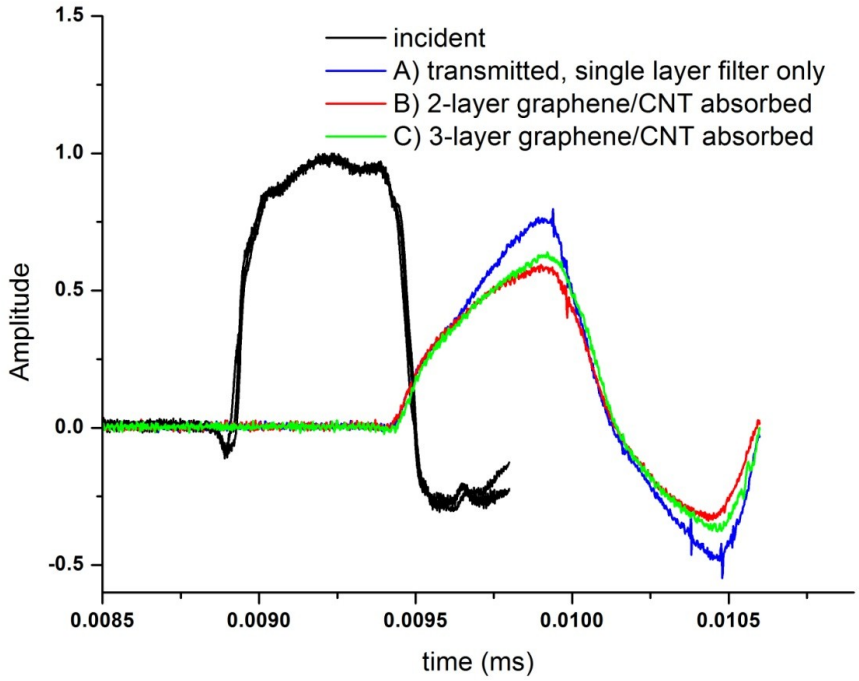


Figure 4.8 Transmitted wave through graphene/CNT absorbed glass microfiber filters. A) Single layer glass microfiber filter. B) 2 layers of graphene/CNT absorbed filters stacking upon each other. C) 3 layers of graphene/CNT absorbed filters stacking upon each other. In each cases, only one glass microfiber filter was used. The filter was exfoliated into two pieces and three pieces in B) and C), respectively, before soaking into graphene oxide/CNT suspensions. Due to limited width of the incident square wave, the actually delay of energy cannot be obtained. Supposing same maximum pressure for all three cases, the delay effect of energy will be B) > C) > A).

Figure 4.8 illustrates the transmitted wave through three different porous media. Surprisingly, the energy delay in pure glass microfiber filter is fairly good. It is suspected that a good amount of gas bubbles still get trapped inside the filter paper even though the filter paper is hydrophilic. In the case of graphene/CNT absorbed filter layers, significant improvement over pure glass microfiber filter is observed assuming equal maximum wave pressure in all three cases. The improvement on the energy delay is resulted from extra water bubble layer trapped in the graphene/CNT surface.

4.3.3 Porous copper thin film

While structured materials formed by packing graphene sheets into filter paper pores show some improvement on the shockwave energy delay, the packing process is very hard to control and the graphene sheets may relocate under the shockwave. Here, an even more stable porous copper thin film is fabricated, which is very promising for long-term use for shockwave delay.

Figure 4.9 shows the morphologies of the thin film during the fabrication process. The drop-casted PS colloidal template (white arena in Figure 4.9A) is thinner in center and thicker at the edge due to the relocation of microspheres during the drying process. A uniform copper layer can be seen beneath the colloidal template surface (Figure B). After the removal of PS microspheres, free standing porous copper thin film can be peeled off from the Si wafer in water. The copper thin film has a thickness of 43 um including a thin layer of Si from the substrate.

SEM images of the porous copper thin film (Figure 4.10A) show randomly distributed pores due to randomly packed PS colloids. A closer look of the porous structure (Figure 4.10B) indicates that those round pores are connect by narrow necks (small dark cicles) and ideally the

whole thin film is a connected network. The as-prepared sample will be further processed under hydrophobic treatment and tested using our Kolsky bar compression test setup.

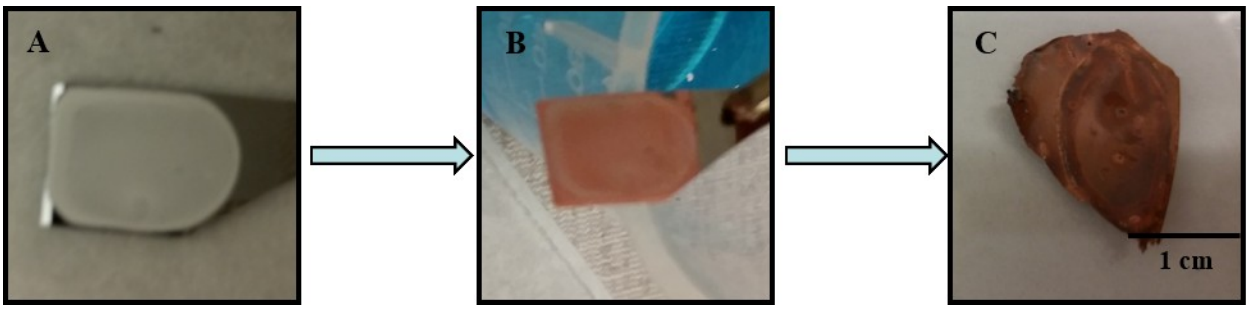


Figure 4.9 Fabrication process of porous copper thin film. A) Polystyrene colloidal template was obtained by drop-casting polystyrene microsphere aqueous suspension on Au-coated Si wafer and drying in air. B) The voids between packed microspheres are filled with copper by electrodeposition. C) 43 um thick free standing porous copper thin film was removed from the Si wafer by water-assisted peeling off technique.

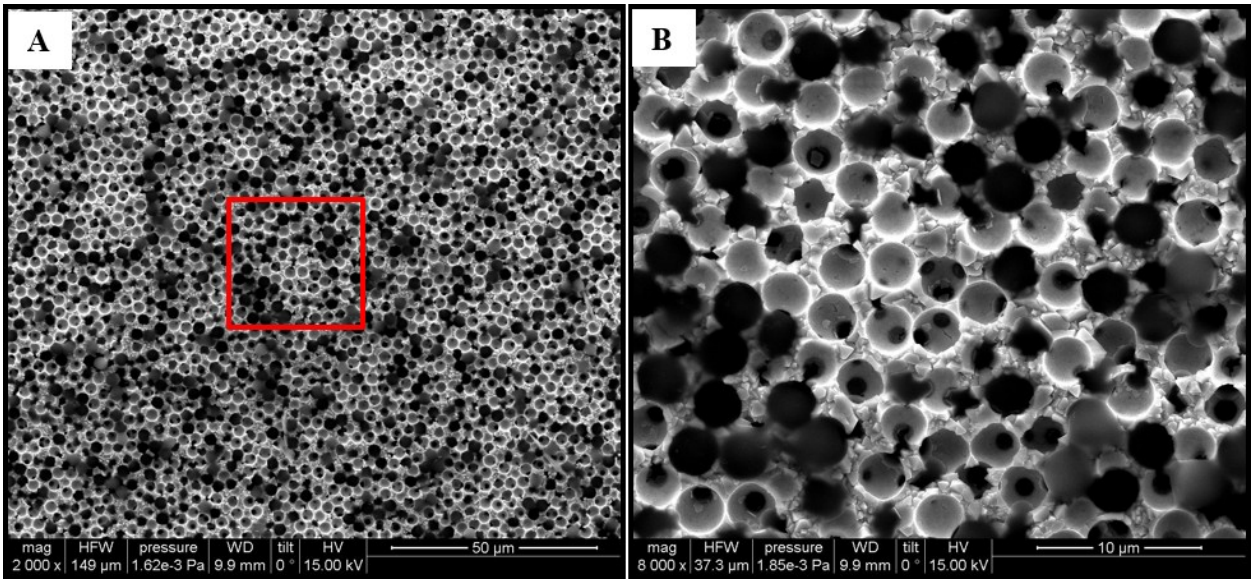


Figure 4.10 SEM images of porous copper thin film. A) Removal of randomly packed colloids leads to randomly distributed round pores. B) A zoom-in view of the marked arena in Panel A shows connected pore structure. The white pores lie on the top layer and dark area indicates lower-layer pores. These pores are connected by narrow necks.

4.4 Summary and future work

In this chapter, free-standing ultra flyweight aerogels with controlled density were fabricated using freeze-drying. In as-prepared UFAs, giant graphene sheets are interconnected via twisting while CNTs serve as ribs. The extremely high porosity and hydrophobicity enables UFA an excellent media to trap air, which has the potential application in energy delay of shock wave. However, the aerogel itself is too fragile. Attempts have been made to fabricate much more robust structured materials by assembling the porous carbon materials into stable porous media such as filter papers. Higher energy delay was observed by simply absorbing of graphene/CNT onto glass microfiber filters. Furthermore, more stable porous copper was fabricated and characterized using SEM. Hydrophobic treatment and Kolsky bar tests will be carried next to study the delay effect.

In shock wave migration, energy damping/dissipation is also very important. A temperature gauge can be installed to track the temperature fluctuation in the cup during Kolsky bar compression test. In order to fully understand the mechanism of air bubbles in energy delay of shock wave, quantitative measurements of trapped air is required.

References:

[4.1] M. F. Ashby, A. G. Evans, N. A. Fleck, et al, *Metal foams, a design guide*. Butterwort-Heinemann, 2000.

[4.2] L. Gibson, M. F. Ashby, *Cellular solids: structures and properties*. Cambridge University Press, 1988.

[4.3] A. G. Evans, J. W. Hutchinson, N. A. Fleck, et al, *Prog. Mater. Sci.* **46**, 309 (2001).

[4.4] L. F. Wang, J. Lau, E. L. Thomas, and M. C. Boyce*, Adv. Mater.* **23**, 1524–1529 (2011).

[4.5] B. N. Cox, N. Sridhar, J. B. Davis, X.-Y. Gong, and F. W. Zok, *Acta Mater.* **48**, 755–766 (2000).

[4.6] D. D. Dubey and A. J. Vizzini, *J. Compos. Mater.* **32**, 158–176 (1998).

[4.7] H. N. G. Wadley, K. P. Dharmasena, M. Y. He, et al, Evans AG, et al, *Prog. Intl. J Impact Engineering* **37**, 317 (2010).

[4.8] B. Xu, X. Chen, W. Lu, et al, *Appl. Phys. Lett.* **104**, 203107 (2014).

[4.9] A. Courtney and M. Courtney, *Med. Hypotheses* **72**, 76–83 (2009).

[4.10] M. Nyein, A. M. Jason, L. Yu, et al, *Proc. Natl. Acad. Sci. U.S.A.* **107**, 20703–20708 (2010).

[4.11] M. W. Seitz and B. W. Skews, *Shock Waves* **15**, 177 (2006).

[4.12] M. W. Seitz and B. W. Skews, *Proceedings of the 20th International Symposium on Shock Waves*, Pasadena, California, July 1995, World Scientific, Singapore; River Edge, NJ, 1996.

[4. 13] 12J. G. M. van der Grinten, *An Experimental Study Of Shock-Induced Wave Propagation In Dry Water-Saturated, And Partially Saturated Porous Media*, Ph.D. thesis, Eindhoven University of Technology, 1987.

[4.14] G. J. Ball, B. P. Howell1, T. G. Leighton, and M. J. Schofield, *Shock Waves* **10**, 265 (2000).

[4.15] A. Bagabir and D. Drikakis, *Comput. Methods Appl. Mech. Eng.* **193**, 4675 (2004).

[4.16] D. C. Marcano, D. V. Kosynkin, J. M. Berlin, .etc, *ACS Nano* **4(8)**, 4806 (2010)

[4.17] C. Gao, C. Vo, Y. Jin, et al, *Macromolecules* **38**, 8634 (2005).

[4.18] H. Sun, Z. Xu, and C. Gao, *Adv. Mater.* **25**, 2554–2560 (2013).

[4.19] H. Kolsky, *Prec. Phys. Soc. London*. **62**, 676-700 (1949).

[4.20] S. Gill, "Helmet-to-helmet Hypocrisy: NFL, NCAA Blame Football Players - When the Problem is Football Programs", *New York Daily News*, 2010.

[4.21] P. Satterthwaite, R. Norton, P. Larmer, and E. Robinson, *British Journal of Sports Medicine* **33,** 22 (1999).

[4.22] H. Zhang, X. Yu, and P. V. Braun, *Nat. Nanotechnol.* **6**, 277 (2011).

[4.23] M. Nienaber, J. S. Lee, R. Feng, and J. Y. Lim, *J. Vis. Exp.* **56**, 2723 (2011).

CHAPTER 5

SOLIDIFICATION OF SUSPENDED COLLOIDS AT NONPLANAR INTERFACE

5.1 Introduction

Phase separation commonly happens in the solidification process of a mixture of several components. This phenomenon has been utilized to fabricate a rich variety of porous structures ranging from nanometer scale to micrometers. For nanometer features, phase separation of atomistic scale components can be adequately driven by diffusion. For example, ultrahigh-density sub-10 nm Co nanowire arrays are fabricated via single target sputtering deposition,[5.1] where diffusion exists throughout a few atomistic layers near the surface. When dealing with micrometer scale components, diffusion is orders of magnitude weaker and the phase separation is mostly driven by a freezing interface of a solvent. In this process, colloidal particles that are dispersed can be either rejected or engulfed by a growing solid-liquid interface. The morphologies of resulting porous materials are therefore dependent on processing conditions, chemistry of the system, and presence of impurities. With this recipe, many polymeric scaffolds[5.2-5.9] or porous ceramics[5.10-5.19] are fabricated as medical implants or tissue regeneration platforms. Instead of delivering a purely random structure, highly ordered polymer fibers with a high aspect ratio (length : diameter > 1000) and smooth surfaces can sometimes be received.[5.20] Clearly, complex nature of the solidification process makes these final constructions highly versatile.

There are a number of experimental and theoretical approaches to reveal the solidification of colloidal suspensions under a unidirectional thermal gradient. To name a few, Deville investigated the behavior of alumina particles during solidification of colloidal suspensions with

an in-situ X-ray tomography, and concluded that particles are redistributed by a direct interaction with the moving solidification interface.[5.21,5.22] Peppin built a continuum model of a unidirectional solidification in a suspension of hard-sphere colloids and studied the volume fraction and temperature profiles ahead of a planar solidification front.[5.23] Numerical simulations have been carried by Garvin to study the interaction of a solidification front with an embedded particle,[5.24,5.25] revealing a critical velocity that is affected by the force expression used in the model. Barr used molecular dynamics (MD) simulations to explore the effect of the ice front velocity on the structure of porous materials with a pre-defined solidification interface.[5.26] While all of these works are inspiring, a self-evolving nonplanar solidification interface in a freezing of colloidal suspension is never introduced, not even mentioning the mutual influence between concentrated or discrete particles and the nonplanar solidification interface.

To partially fill such a gap, a two dimensional numerical model is developed in this paper to simulate structure growth dynamics from a unidirectional freezing or solidification process. For the first time, the phase change of the solvent and the motion of many colloidal particles are fully coupled and their mutual influences are presented. Resulting nonplanar solidification interfaces are described and corresponding particle concentration profiles are illustrated. Furthermore, the dynamics of how growing interface surround and encapsulate moving particles are described to a very fine, particle level. In order to view simulation data and capture highly intermittent transient phenomena, we also develop an in-situ visualization that tightly couples with our numerical simulation by programming both the simulation module and the visualization module on high performance GPU devices.

5.2 Methodologies

5.2.1 Simulation system setup

Figure 5.1 shows the sketch of the physical model used in our simulation. This directional freezing stage has been used in our or others experiments to observe the evolution of the solid-liquid interface.[5.27] The cold (green, T_L) and hot blocks (red, T_H), with the temperature satisfying $T_L < T_m < T_H$ (T_m = the melting point of the suspension), are in good thermal conduction with the growth cell (rectangle). The cold/hot blocks move at a constant speed of v_s, which is the equilibrium freezing speed of the process. As a result, a thermal gradient is created between these two blocks and a stable solidification interface is developed. The solid-liquid interface may push or engulf colloidal particles (white dots) depending on freezing conditions.

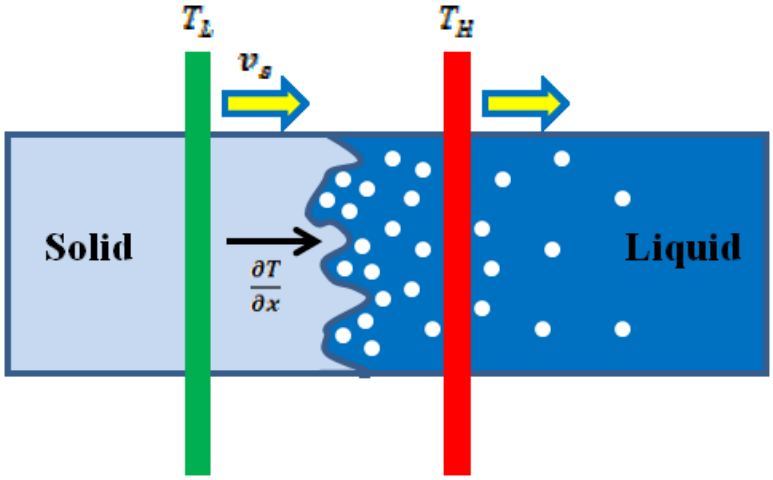


Figure 5.1 Sketch of our physical model in a unidirectional freezing. Green and red bars represent cold and hot blocks with the temperature satisfying $T_L < T_m < T_H$, where T_m is the melting point of the suspension. The solidification interface may push or engulf colloidal particles (white dots) in liquid depending on the freezing conditions.

5.2.2 Solidification interface via phase-field method

The above 2-D solidification is modeled using the phase-field method (Figure 5.2) based on Kobayashi formulation.[5.28] The advantage of this continuum approach over other methods

(such as level set based method[5.29]) dealing with a moving boundary lies in that the interface is implicitly expressed. Therefore, it is well-suited to simulate moving boundaries involving splitting and merging. The Kobayashi formulation incorporates the effect of anisotropy and supercooling, delivering the growth of cellular and dendritic crystals (Figure 5.2(B&C)). The governing equations[5.28] for the phase-field are given as

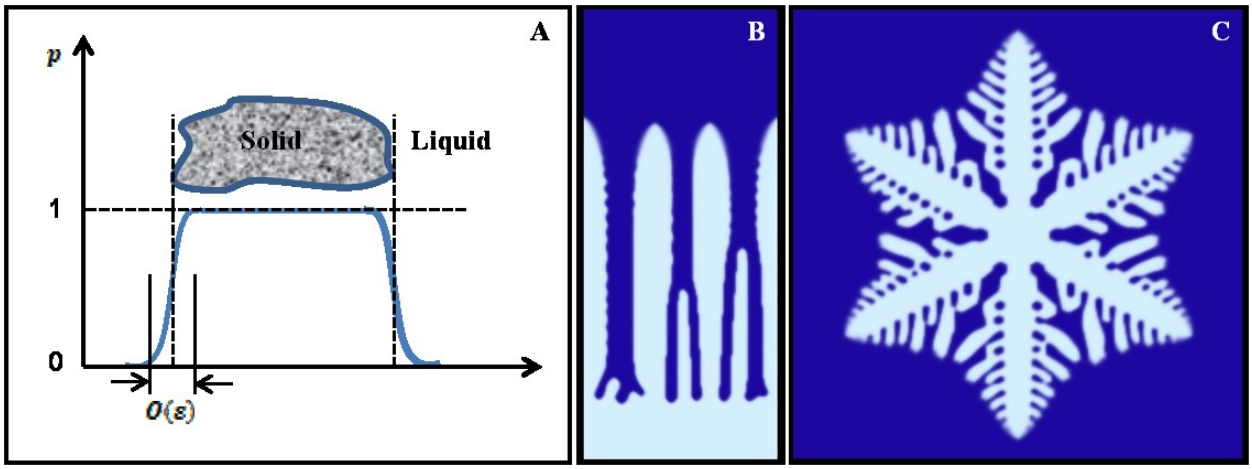


Figure 5.2 Phase field method for solidification simulation of a pure solvent. A) A phase order parameter p is defined, with 1 representing solid and 0 as liquid. Parameter ε determines the interface thickness, within which $p \in [0, 1]$. Morphologies grown in supercooled condition are shown in B) cellular structure from unidirectional growth, and C) snow-flake dendritic structure from the nucleation at center. Blue and white areas represent liquid and solid, respectively. Panel A is modified from Reference 5.28.

$$\tau \frac{\partial p}{\partial t} = -\frac{\partial}{\partial x}\left(\varepsilon\varepsilon' \frac{\partial p}{\partial y}\right) + \frac{\partial}{\partial y}\left(\varepsilon\varepsilon' \frac{\partial p}{\partial x}\right) + \nabla \cdot (\varepsilon^2 \nabla p) + p(1-p)(p - \tfrac{1}{2} + m), \tag{5.1}$$

$$\frac{\partial T}{\partial t} = \nabla^2 T + K \frac{\partial p}{\partial t}, \tag{5.2}$$

which describe the interplay between the order parameter p ($p \in [0, 1]$) and temperature field T. K is the dimensionless latent heat that is proportional to latent heat and inversely proportional to the strength of cooling; ε is a small parameter that determines the thickness of the interface; and m is a function of T indicating the driving force of interfacial motion from supercooling. Taking

$\varepsilon = \bar{\varepsilon}\sigma(\theta)$, where $\bar{\varepsilon}$ is the mean value of ε and $\sigma(\theta)$ represents anisotropy, the time evolving equations lead to an implicit interface effect at the singular limit

$$\tau V = \sqrt{2}m(T)\sigma(\theta) - \left[\sigma(\theta)^2 + \left(\frac{1}{2}\sigma(\theta)^2\right)''\right]k\bar{\varepsilon}, \tag{5.3}$$

where $\sigma(\theta)$ and $m(T)$ are specified as

$$\sigma(\theta) = 1 + \delta[1 + \cos(j\theta)], \tag{5.4}$$

$$m(T) = \left(\frac{\alpha}{\pi}\right)\tan[\gamma(T_m - T)], \tag{5.5}$$

with V as the growth velocity of interface in normal direction, k depicting interface curvature, m representing the supercooling at interface, δ standing for the strength of anisotropy, and j as the mode number of anisotropy.

In our simulation model, external heat sources are used (cold/hot blocks in Figure 5.1). In this case, Equation 5.2 is modified as below to include the temperature control from moving cold/hot blocks.

$$\frac{\partial T}{\partial t} = \mathrm{D}_T\nabla^2 T + K\frac{\partial p}{\partial t} - v_s\frac{T_H - T_L}{d_{cell}}, \tag{5.6}$$

where T_H and T_L are the temperature of the cold and hot blocks, respectively. d_{cell} is the distance between two blocks and v_s is the sliding speed of two blocks.

The detailed derivation of the Equation 5.1 and 5.3 can be found in Appendix A1

5.2.3 Colloid motion via discrete element method

For simplicity, colloid particles are treated as 2D hard disks and a Discrete Element Method (DEM)[5.30] is employed to deal with particle collisions. DEM is a discrete approach for computing the motion of a large number of isolated particles. In this model, the force between a pair of particles i and j is given as

$$\vec{f}_{ij,particle} = -k_r\left(d - \left|\vec{r}_{ij}\right|\right)\frac{\vec{r}_{ij}}{|\vec{r}_{ij}|} + \mu_d\vec{v}_{ij} + k_t\vec{v}_{ij,t} + \alpha_a\vec{r}_{ij}, \tag{5.7}$$

where four terms on the right side represents repelling, damping, shear, and attractive forces, respectively; d is the sum of radius of particles i and j; $k_r, \mu_d, k_t, \alpha_a$ are parameters to indicate the level of contribution of each term; and $\vec{r}_{ij}$ and $\vec{v}_{ij}$ are relative displacement and velocity vector.

Particle-particle interactions and particle-interface interactions (given in next section) in collision are highly localized, and thus can be accelerated with the aid of laying a uniform particle grid with spacing equal to the diameter of the particle.[5.31] The algorithm is implemented on GPU devices and details are given in section 5.2.4.

5.2.4 Coupling between solidification interface simulation and colloid particle simulation

A fully coupled simulation model, which combines the solidification interface growth and the discrete motion of colloidal particles, is developed for our system. Overall, the growing solid-liquid interface and particle motion affect each other. In particular, the particles are subject to the repelling/drag forces from the solidification interface; in the meantime, the morphology and moving speed of the interface are perturbed by constitutional supercooling that is resulted from the concentration of particles at the solidification front. Since we use different simulation frameworks to couple these two issues, for simplicity, the force of the interface acting on a colloid particle k is treated as the total force of solid phase points near the interface on the particle (Equation 5.8, Figure 5.3A).

$$\vec{f}_{k,interface} = \sum_{r_{jk}<r_c} p_j \vec{f}_{jk}, \tag{5.8}$$

where r_{jk} is the distance between a phase point j and a particle k, r_c is an arbitrary cutoff radius, p_j is the phase order parameter and $\vec{f}_{jk}$ is given in Equation 5.7. A cutoff range r_c is applied to reduce the computational cost, within which neighboring particles in contact with the particle may collide with each other. The DEM model is also applied to deal with this collision.

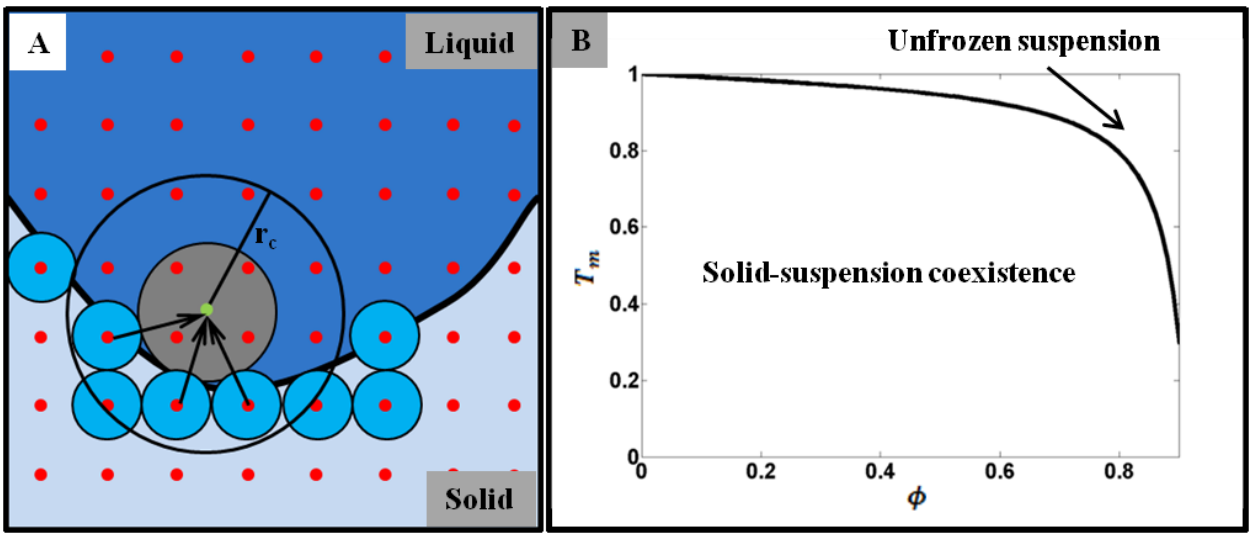


Figure 5.3 Interaction between solid-liquid interface and colloidal particles. A) The interaction of the interface (black line) onto a particle (gray disk) is calculated as the total force of solid phase points near the interface (light blue disks) to the particle. In this case, the solid phase points near the interface are considered to be disks with diameter equal to the phase field (red dots) spacing s. A cutoff range r_c is applied to reduce the computational cost. Only those points falling within the cutoff range may interact with the particle. B) Freezing temperature (T_m) depression versus colloidal particle concentration (ϕ). The particle concentration at a point is calculated as the total area of the disks (2D particles) inside an arbitrary circle with a center located at this point divided by the area of the circle.

Beyond inter-particle processes, the colloid is also subject to a global viscous force resulted from the liquid viscosity.

$$\vec{f}_{k,viscous} = -\mu_v \vec{v}_k, \tag{5.9}$$

As a result, the total force acting on a colloid particle k is the sum of the force from other colloid particles, the force from solid-liquid interface, and the viscous force. Generally, it can be written as

$$\vec{f}_k = \vec{f}_{k,particle} + \vec{f}_{k,inteface} + \vec{f}_{k,viscous} = \sum_{r_{jk}<r_{cut}} \vec{f}_{jk,particle} + \sum_{r_{jk}<r_{cut}} p_j \vec{f}_{jk} - \mu \vec{v}_k, \tag{5.10}$$

On the other hand, earlier studies have shown that the increase of the colloid particle concentration at the solidification front leads to a decrease of freezing temperature.[5.23] In our study, this effect is given as

$$T_m = T_m^0 - \gamma_{depress} tan\ (\omega\phi), \tag{5.11}$$

where T_m is the melting point of the solidification front at a particle concentration of ϕ, T_m^0 is the freezing temperature of pure solvent, and γ_f and ω are fitted parameters. Equation 5.11 yields a similar $T_m - \phi$ curve (Figure 3B) to the one developed by Peppin.[5.23] In a unidirectional freezing of colloidal suspensions, such a depression in freezing temperature may result in the suspension ahead of the solid-liquid interface to be below its freezing point (constitutional supercooling). Constitutional supercooling is closely related to morphological instability, therefore generating a nonplanar interface.[5.35] Certainly, this nonplanar interface will further introduce a localized supercooling due to the particle redistribution. In our 2D model, the particle concentration at a point k is calculated as the total area of the disks (2D particles) inside a circle of a radius of r_{cp} with the center at this point, divided by the area of the circle. Simply, it can be written as

$$\phi_k = \sum_{r_{jk}<r_{cp}} S_j / S_r, \tag{5.12}$$

where S_j represents the area of disk j and S_r is the area of the circle. A typical set of phase field and colloidal particle parameters are listed in Table 5.1 and Table 5.2, respectively.

In order to explore the detailed particle entrapment in a much finer particle scale, much smaller phase field grid spacing is used ($s : r_{cp} = 1 : 5$). In computation, this is equivalent to increasing the particle size while keeping the grid spacing fixed. Modified parameters are shown in Table 5.3 while other parameters can be found in Table 5.1 and Table 5.2.

Table 5.1 Phase field model parameters

α	γ	$\bar{\varepsilon}$	τ	D_T	a	θ_0	T_e	K	δ	j	dt
0.9	10	0.01	0.0003	1.0	0.1	$\pi/2$	1.0	1.0	0.05	4.0	0.0001
grid size		s	T_L	T_L	v_s	d_{LH}					
1024 × 256		0.03	-1.0	2.0	8.0	5.0					

Table 5.2 Colloidal particle parameters

μ_v	k_r	μ_d	k_t	α_a	r_p	$p_{threshold}$	$\gamma_{depress}$	α_{engulf}	w
0.2	500.0	0.02	0.1	0	0.01	0.5	0.5	0.9	1.5

Table 5.3 Parameters for entrapment of colloidal particles at particle scale

Grid size	K	r_p	r_{ct}
512 × 128	0.0	0.15	$1.25 \times (s + r_p)$

Notice that the latent heat from phase change is not considered here ($K = 0$) because its effect on the interface instability is trivial compared to that from local curvature created by the particle surface. The particle densities on phase points inside and outside the particle are set to be 1 and 0, respectively. Applying the particle densities to Equation 5.11 leads to a very low freezing point ($< T_L$) inside the particle and T_m^0 outside the particle. In this case, freezing of the solvent will never penetrate into the particle. On the other hand, while previous technique (Equation 5.8, Figure 5.3A) yields acceptable results for larger scale simulations, it will perform poorly when a low ratio of phase field grid spacing to particle size is used. As such, a Euclidean vector technique is developed to find the collision site of the interface with a particle and DEM model is then applied for the interface-particle collision (Figure 5.4). Particularly, the phase field grid cells representing the interface (interface grid cells) are determined by comparing the phase order parameter p of four vertexes of each grid cell with $p_{threshold}$:

$$\text{phase field grid cell } k \text{ represents } \begin{cases} solid & if\ p > p_{threshold}\ for\ all\ 4\ vertexes, \\ liquid & if\ p < p_{threshold}\ for\ all\ 4\ vertexes, \\ interface & otherwise. \end{cases} \tag{5.13}$$

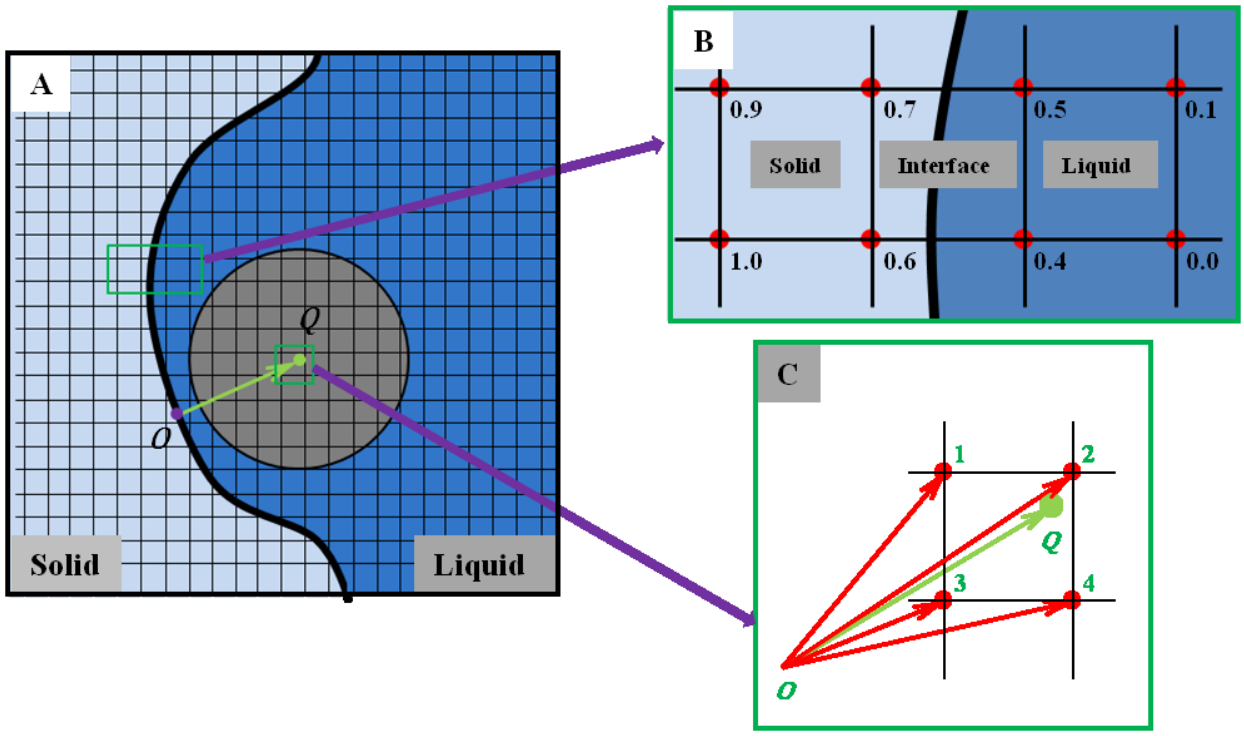


Figure 5.4 Sketch showing the process of determining the collision between a nonplanar solid-liquid interface and a colloid using a Euclidean vector technique. The phase field grid cells representing the interface (interface grid cells) are those passed through by the solid-liquid interface (black curve in Panel A), which are mathematically determined by comparing the phase order parameter p of four vertexes of each grid cell with $p_{threshold}$ (Equation 13). Panel B illustrates an example of determining the interface grid cells using $p_{threshold} = 0.5$. The collision location (purple dot O in Panel A) of the interface with a colloid particle (gray disk with center at Q) is determined by searching the shortest Euclidean vector from each interface grid cell to point Q, which is carried out in the following two steps. First, the shortest Euclidean vector from the interface to each phase field grid vertex is found by looping over all neighbor interface grid cells near the grid vertex. Then the shortest Euclidean vector from the interface to any location Q in the space can be approximately calculated by bilinear interpolation of the shortest Euclidean vector from the interface to four vertexes surrounding this point (Panel C).

An example of determining the interface grid cells is shown in Figure 5.4B using $p_{threshold} = 0.5$. Connecting those interface grid cells yields a good approximation to the real

solid-liquid interface (black curve in Figure 5.4A). Second, the collision location (purple dot O in Figure 5.4A) of the interface with a colloid particle (gray disk with center at Q) is determined by searching the shortest Euclidean vector from each interface grid cell to point Q. A Euclidean vector is defined as a line segment with a definite direction. In order to perform the search efficiently for all colloids located everywhere in the system, the shortest Euclidean vector from the interface to each phase field grid vertex is found by looping over all neighbor interface grid cells near the grid vertex. Then the shortest Euclidean vector from the interface to any location Q in the space can be approximately calculated by bilinear interpolation of the shortest Euclidean vector from the interface to four vertexes surrounding this point (Figure 5.4C). Finally, DEM model is also applied for the collision between a particle located at Q and an interface with the collision location at O.

We used two different particle trapping criteria throughout the simulation, with Equation 5.14 for the nonplanar solid-liquid interface in Sections 5.3.1 and 5.3.2, and Equation 5.15 for the detailed particle entrapment at particle scale for later sections. Equation 5.14 is valid even with a large number of particles (hundreds of thousands), but Equation 5.15 for hundreds of them at particle scale. Particularly, for the former case, a colloid particle is considered as engulfed by the interface (frozen) if any solid phase point j ($p_j > p_{threshold}$) is found within a certain distance from the colloidal particle k, which is given as

$$\exists k: r_{jk} < \alpha_{engulf}(s + r_p), \tag{5.14}$$

where r_{jk} is the distance between a solid phase point j and a particle k, s represents phase field spacing, r_p stands for particle radius, and α_{engulf} is an arbitrary parameter between 0 and 1. For the latter case with hundreds of particles, Equation 5.15 provides a more accurate entrapment criterion, where a particle is treated as encapsulated by the interface if half of the particle surface

is covered by solid solvent ($p > p_{threshold}$). Mathematically, the entrapment criteria for a particle k is expressed as,

$$\frac{c(\{j:r_p<r_{jk}<r_{ct}, p_j>p_{threshold}\})}{c(\{j:r_p<r_{jk}<r_{ct}\})} > 0.5, \tag{5.15}$$

where $c()$ counts the number of elements in the set and j represents a phase point, and r_{ct} is a cutoff range usually take as $1.25 \times (s + r_p)$.

5.2.5 GPU implementation and in-situ visualization of the solidification process

Solidification of colloidal suspension is an extremely complex problem because of a large number of parameters involved, including colloid particle properties, anisotropy, heat conduction, and many others. Here, we develop an interactive simulation scheme (Figure 5.5) that enables real-time simulation and visualization of solid-liquid interface, temperature, and colloid particle concentration profile. The visualization options can be selected to investigate the local properties, such as rotation, zoom, point or surface representation. The real-time adjustment of parameters through a user-friendly interface allows a fast exploration of the effect of aforementioned parameters.

At each time step, the phase field including the solid-liquid interface information and the particle velocities and positions are updated (Figure 5.6). At the end of each step, these data are rendered and illustrated on the display. In order to achieve real-time visualization of the growth process, the computation capacity of at least a few frames per second is a must. We code the solidification interface growth and discrete particle motions on GPU devices using CUDA[5.31], and use OpenGL[5.32] for the in-situ visualization. GPUs are optimized for taking huge batches of data and performing the same operation over and over very quickly, which are highly suitable for our repeated calculation at each time step.

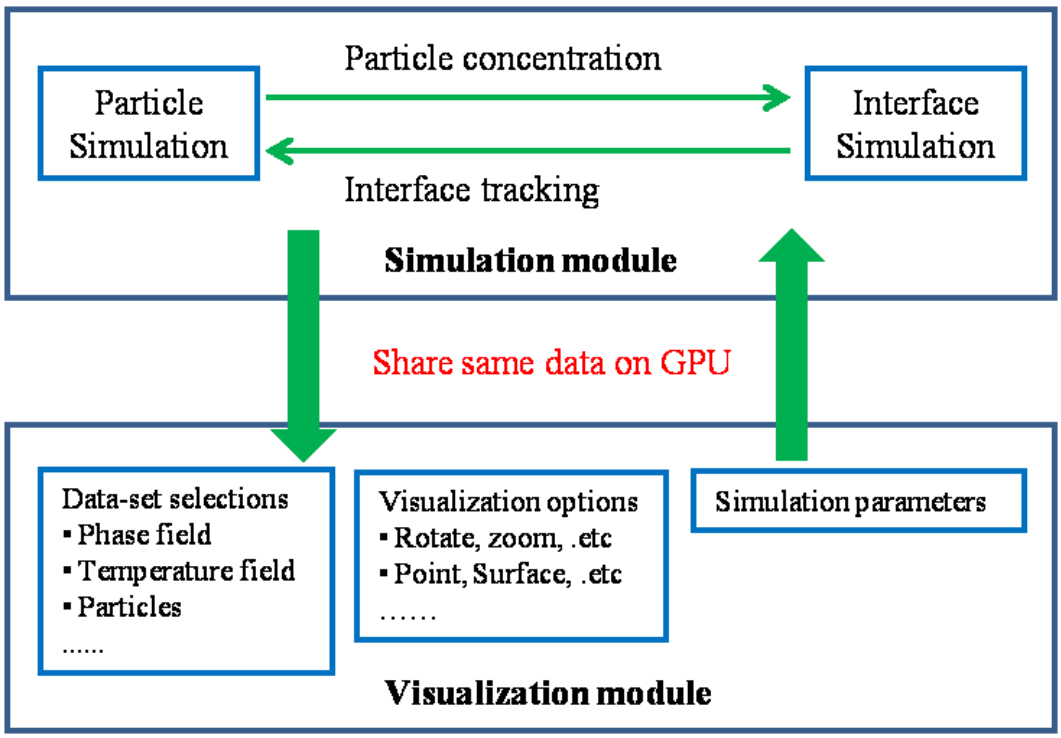


Figure 5.5 Simulation and visualization scheme on GPU devices. Both solidification interface growth and discrete particle motions are coded using CUDA[5.31] on GPU devices. Both these modules are further combined with a visualization module to access the same data storage, enabling display of a real-time status using OpenGL[5.32]. A user friendly interface is created to monitor the display and simulation parameters.

Solving Equation 5.1 and Equation 5.6 on a uniform grid at each time step using a simple explicit scheme simulates the solidification interface growth. Because of the high localization of the particle-particle interaction and interface-particle interaction, the particle motions on GPUs can be accelerated by building a particle grid and applying the DEM method for particle collisions.[5.31]

In our simulation, we built a uniform grid with the cell size equal to the size of a particle. In this case, each particle can only cover a limited number of grid cells (4 in 2D), and there is a fixed upper limit on the number of particles per grid cell (4 in 2D) assuming no inter-penetration between particles. To process the collisions for a particle, only the particles in the neighboring

cells (3 × 3 in 2D) are examined to see if they are in touch with the particle. In order to process the collisions for all particles, the sorted list of particles in each cell is generated from scratch at each time step. Figure 5.7 demonstrates the creation of a 4 × 4 grid structure with 7 particles using a sorting approach. After the grid structure is finished, it is used to accelerate the particle collisions using the DEM method.

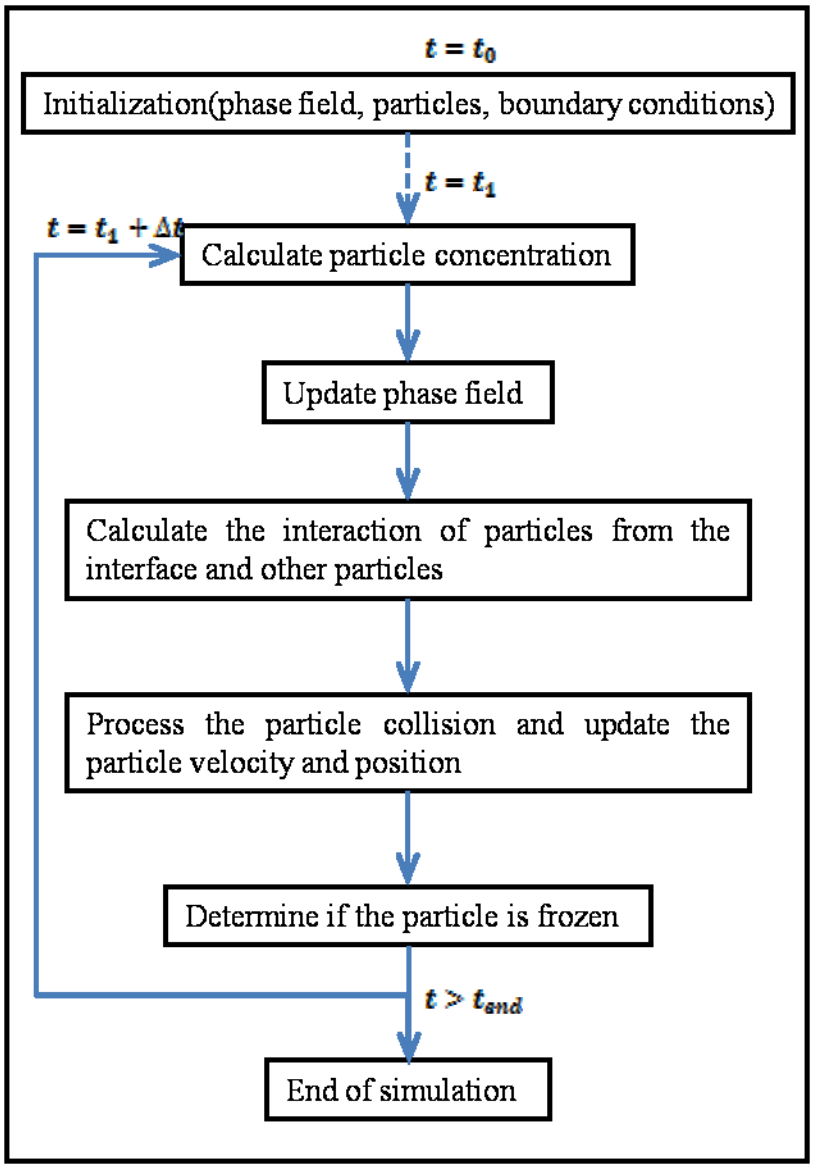
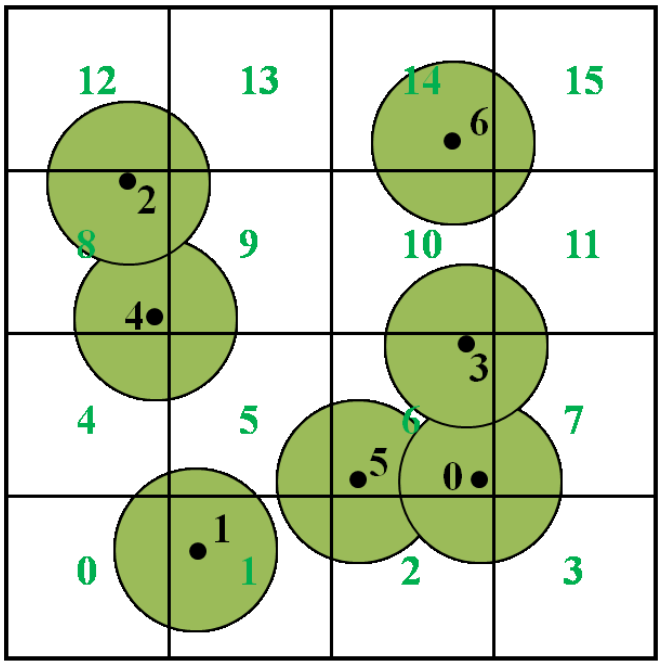


Figure 5.6 Flow chart of the simulation steps. Phase field including the solid-liquid interface information and the particle positions are updated every step.



Index	Unsorted list (cell id, particle id)	List sorted by cell id	Cell start
0	(6, 0)	(1, 1)	
1	(1, 1)	(6, 0)	0
2	(8, 2)	(6, 3)	
3	(6, 3)	(6, 5)	
4	(8, 4)	(8, 2)	
5	(6, 5)	(8, 4)	
6	(14, 6)	(14, 6)	1
7			
8			4
9			
10			
11			
12			
13			
14			6
15			

Figure 5.7 Example of creating of a 4 × 4 grid structure (left panel) with 7 particles (green disks) using a sorting approach. The algorithm consists of several kernel functions. The first kernel calculates a hash value for each particle based on its cell id. In our case, the linear cell id is used as hash, and the results are stored to an array in global memory as a uint2 pair (cell hash, particle id, column 2 of the table on the right). Then a second kernel sorts the particles based on their hash values (column 3). The sorting is performed using the fast radix sort provided by the CUDPP library. In order to be able to access the particle list in any given cell, the start location of this cell in the sorted list has to be stored, which is achieved by the third kernel. This kernel uses a thread per particle and compares the cell index of the current particle with the cell index of the previous particle in the sorted list (column 3). If the index is different, a non-empty cell is started, and the start address is written to another array using a scattered write (column 4). The index of the end of each cell can be found and recorded in a similar way. After the grid structure is finished, it can be used to accelerate the particle collisions using a DEM method. Left panel is modified from Reference 5.31.

Table 5.4 Performance test of the GPU code

Phase field grid size	Execution time / step (s)		Speedup
	Opteron 6272 2.1G Hz CPU	Nvdia GTX 470 GPU	
128 × 512	0.1682	0.0167	10.07
256 × 1024	0.6752	0.0167	40.43
512 × 2048	2.6908	0.0175	153.76

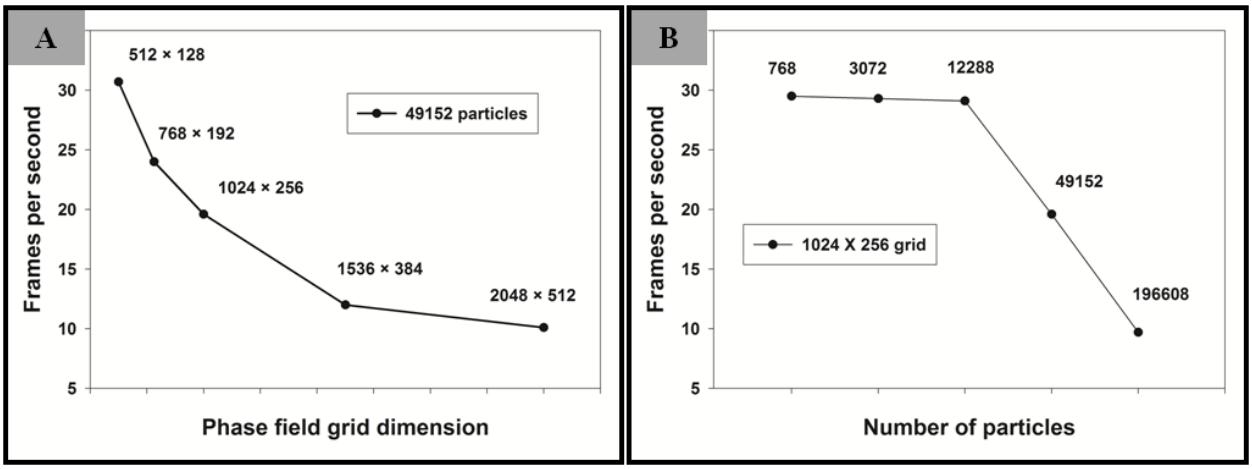


Figure 5.8 Frames per second (FPS) of the in-situ visualization at different system size. At least 10 FPS is required for a comfortable real-time visualization. The FPS drops with the increase of the system size: A) the number of particles is fixed and B) phase field grid size is fixed.

Our performance test shows that the GPU implementation of the phase field simulation is over 100 times faster than the CPU counterpart (Table 5.4). The performance test was carried out on the solidification of a pure solvent without any particle involved. The algorithm for particle simulation on the GPU is unique and cannot be reproduced on CPU. Therefore, no direct comparison of the performance on particle simulation was made but high speedup is also expected. For in-situ visualization, at least 10 frames per second (FPS) can be maintained for a good quality illustration of a system with millions of phase field points and hundreds of

thousands of colloidal particles even on a low-end GPU device (Nvidia GTX 470). The FPS drops with the increase of the system size, either the number of particles or phase field grid size (Figure 5.8). Demo video of the interactive simulation framework can be obtained by contacting the author. The source code for the GPU implementation is attached in Appendix A2.

5.3 Results and Discussion

5.3.1 Instability and morphology of 2D solid-liquid interface

In simulation of unidirectional freezing of a supercooled pure fluid, the surface instability can be achieved by introduction of latent heat along the moving interface. The latent heat from the phase change could lead to a negative temperature gradient (thermal supercooling) ahead of the interface even in unidirectional freezing. In this case, cellular and dendritic structures are observed in the simulation box (Figure 5.1(B&C)), which are similar to the structures developed by Kobayashi.[5.28] Detailed information of the simulation parameters and interface morphologies at different latent heat parameters can be found in Table 5.1 and Figure 5.9, respectively. The flat surface is stable at K = 0.2 and a little bit destabilized at K = 0.3. A weak cellular structure is observed for K = 0.5. The cellular tips become sharper with the increase of K value (Figure 5.9(D, E, and F)). At K = 3.0, the edge of the tips appears fuzzy, in other words, the solid-liquid interface becomes thick, which brings in large numerical errors and violates the assumption of thin interface made in Figure 5.2A. K = 1.0 is chosen in later simulations unless stated otherwise.

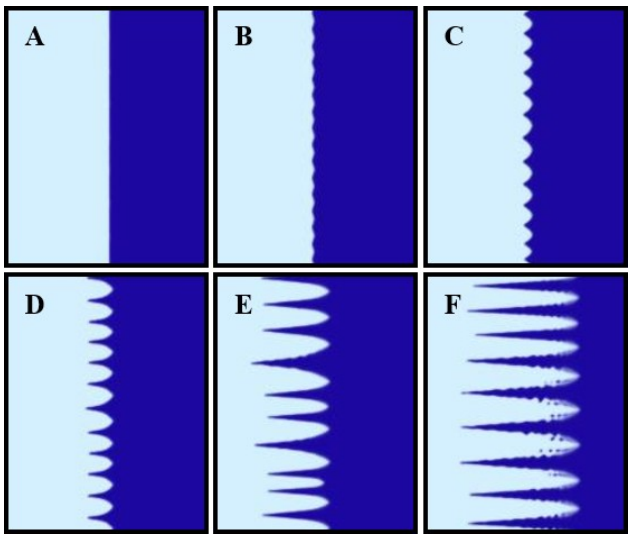


Figure 5.9 Morphology of the equilibrium solidification interface using different dimensionless latent heat parameter K (Equation 2): A) 0.2; B) 0.3; C) 0.5; D) 1.0; E) 2.0; F) 3.0. K is proportional to latent heat and inversely proportional to strength of cooling. White area and blue arena represent solid and liquid, respectively. Other parameters used in the phase field model are shown in Table S1. The flat surface is stable at $K = 0.2$ (A) and a little bit destabilized at $K = 0.3$ (B). A weak cellular structure is observed for $K = 0.5$ (C). The cellular tips become sharper with the increase of K value (D, E, and F). At $K = 3.0$, the edge of the tips appears fuzzy, in other words, the solid-liquid interface becomes thick, which brings in large numerical errors and violates the assumption of thin interface made in Figure S1A. $K = 1.0$ is chosen in later simulations unless stated otherwise.

In solidification of colloid suspension, the surface instability is much more complex. The nonlinear dependence of freezing temperature on colloid concentration could lower the actual freezing temperature of the suspension ahead of the solid-liquid interface and may lead to constitutional supercooling (particle in solvent induces a positive freezing point gradient). In this case, interface instability will occur and the interface tends to deform into the supercooled region, generating a nonplanar interface. We observed this interface instability with the formation of rough surfaces (Figure 5.10A) (particle simulation parameters in Table 5.2). However, resulting solid textures are small and highly disordered compared to that from thermal

supercooling only (Figure 5.10B, latent heat induces a negative temperature gradient). The combined effect of both constitutional supercooling and thermal supercooling is shown in Figure 5.10C, which possesses very similar pattern found in experiment.[5.27] A negative temperature gradient is also observed in the temperature distribution profile. Therefore, we conclude that both constitutional and thermal supercooling contribute to the development of a nonplanar interface. The corresponding phase field distribution and particle concentration distribution are also shown in Figure 5.10.

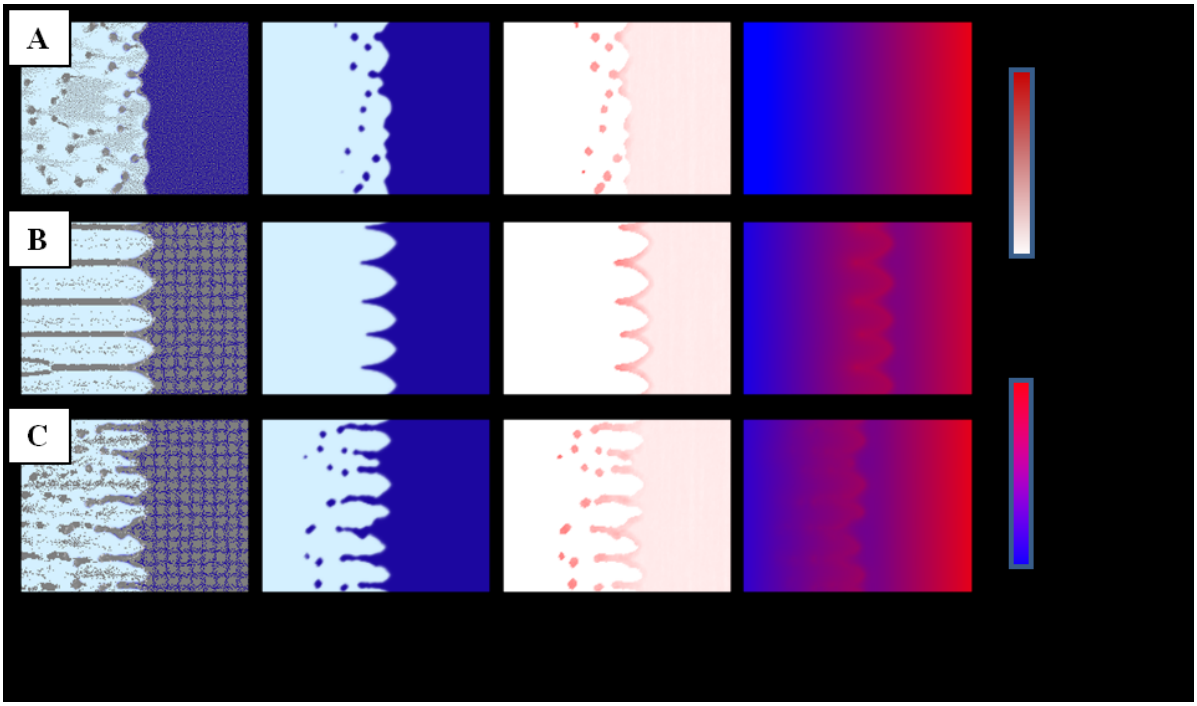


Figure 5.10 Morphologies of the solidification interface in colloid suspension resulted from A) constitutional supercooling only, B) thermal supercooling only, and C) combined thermal and constitutional supercooling effect. Four columns correspond to: 1) snapshot showing phase field and colloidal particles (white - solid, blue - liquid, gray - colloidal particles), 2) phase field showing clear solid-liquid interface, 3) particle density profile, 4) temperature field. Thermal supercooling leads to highly ordered and oriented solid tips (Panel B), while constitutional supercooling leads to more disordered but smoother solid-liquid interface (Panel A). The combined effect of thermal and constitutional supercooling leads to the solidification interface in Panel C, which is commonly observed in experiments.

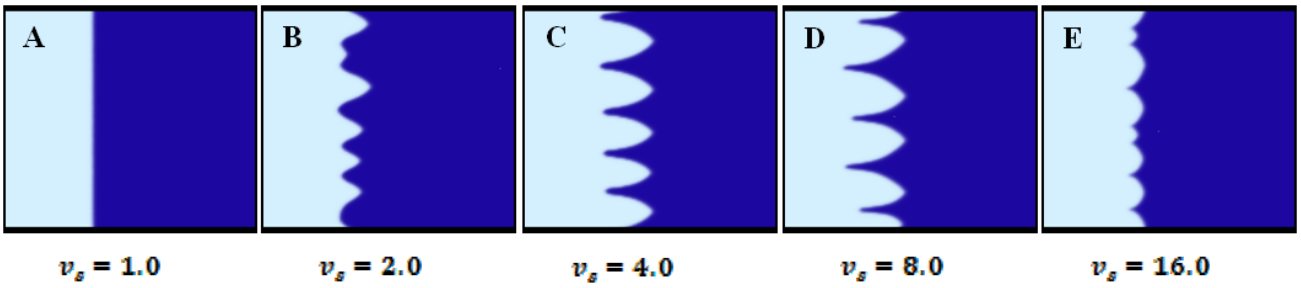


Figure 5.11 Interface morphology under different pulling speed, which equals to the freezing speed v_s at equilibrium: A) 1.0; B) 2.0; C) 4.0; D) 8.0; E) 16.0. A transition from planar to nonplanar interface was observed between v_s = 1.0 and v_s = 2.0. Well-ordered solid tips were observed when the freezing speed was further increased (Panel C&D). At extreme high pulling speed (Panel E), the solidification interface cannot keep up with the pulling speed, which leads to a solid-liquid interface with short tips very close to the cold block.

The solid-liquid interface is also affected by the freezing speed, which equals to the pulling speed of the cold/hot blocks at equilibrium (Figure 5.11). At very low freezing speed (v_s = 1.0), the planar interface is stable (Figure 5.11A). The interface starts to destabilize when the freezing speed is increased (Figure 5.11B) where the solid tips become sharper, eventually leading to the formation of well-ordered tips (Figure 5.11(C&D)). This trend is in good agreement with the experimental observations in unidirectional solidification of montmorillonite clay.[5.23] At an extremely high pulling speed (v_s = 16.0, Figure 5.11E), solidification interface cannot keep up with the pulling speed, which leads to a solid-liquid interface with short tips very close to the cold block.

5.3.2 Discrete particle motion at solidification front

In solidification of a colloidal suspension, the movement of solid-liquid interface leads to motion and the redistribution of colloidal particles. Figure 5.12 illustrates the colloid particle distribution near the solidification interface for a planar and a fiber-like pattern. When the freezing speed is low (v_s = 1.0), all particles are rejected by a flat surface, with concentration

maximum occurring near the solidification interface but quickly decreasing to match the bulk colloid concentration at location not far away from the interface (Figure 5.12B, along the purple line shown in Figure 5.12A). When the freezing speed is high enough (v_s = 8.0), the solidification interface becomes unstable and a finger-like feature is observed (Figure 5.12C). In this case, colloid particles are repelled and engulfed between the solid fingers, eventually leads to the formation of fibers. A cursor plot confirms that the particles are heavily concentrated between the fingers, with few of them in front of each finger (Figure 5.12D, along purple and green lines in Figure 5.12C).

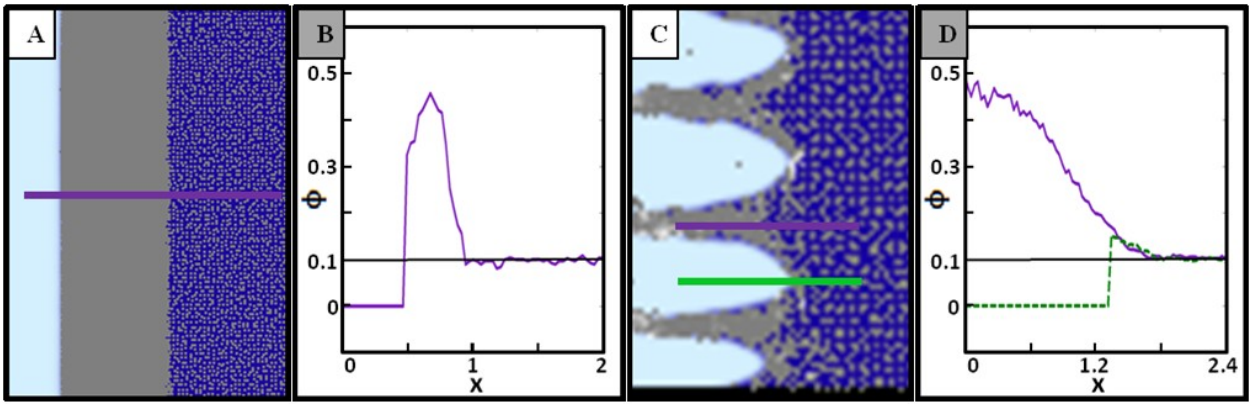


Figure 5.12 Colloid particle concentration in the solidification front for A) a flat surface, and C) a finger-like surface. Gray dots stand for colloid particles, white area represents solid and blue arena represents solvent. When the freezing speed is low (v_s = 1.0), all particles are repelled by the flat surface. Colloid particles are concentrated near the solidification surface and the density quickly reduces to colloid suspension concentration when away from the interface (Figure B, along the purple line shown in Panel A). When the freezing speed gets high (v_s = 8.0), the solidification interface becomes unstable and a finger-like surface maybe observed (Panel C). In this case, colloid particles are repelled and engulfed between the solid fingers, eventually leads to the formation of fibers. Particles are concentrated between the solid fingers and few particles are found within the solid fingers (Panel D, along purple and green lines in Panel C).

The maximum particle concentration achieved in both cases is around 0.5, which is below the random close-packing density of 0.84 for 2D disks[5.33], indicating a low crystallinity in the particle packing.

5.3.3 Entrapment of particles at solid-liquid interface

In previous sections, we showed that colloidal particles can be either rejected or engulfed by the growing solid-liquid interface depending on the freezing conditions. Now we can zoom in at the finger gaps in Figure 5.12 and observe the details of particle motions (parameters shown in Table 5.3, supporting information). Particularly, the interactions (k_r , Equation 5.7) between the interface and colloidal particles play a very important role. When the repulsive force between the interface and the particle is relatively weak (k_r = 100), growing solid bends and surrounds the particle surface even at low particle density, and eventually engulfed and trapped the particles (Figure 5.13A). On the other hand, if the repelling force from the interface is strong (k_r = 500), those colloid particles will be pushed away by the interface without being trapped (Figure 5.13B). In either case, the loosely packed particles will eventually be condensed. However, if those loosely packed particles in a suspension are confined in a rough and narrow gap like in Figure 5.12C, not only will the particles be condensed after the solidification but they will also be penetrated by a growing front of frozen solid. In our simulation, this is revealed at a rather late stage of the freezing process, where the situation is more or less like having a rigid wall on the far right (Figure 5.13C). As a result, newly developed frozen solid starts to enter the pores between the particles, without breaking the particles apart. However, we also notice that the liquid in small pores are harder to be frozen than that in large pores, which is in good agreement with the description of ice-entry temperature in Peppin's work.[5.23]

If we relax the aforementioned boundary on the far right, the entrapment of particles in frozen solid can also occur at a faster freezing speed (k_r is fixed to 500). At a relatively slow cooling speed (v_s ~ 8.0), since those particles are free to move, no frozen-solid-entering-pore will occur even if the suspension is highly concentrated (Figure 5.14). However, the solidification interface becomes unstable when the freezing speed is increased. A transition from fully repelling to entrapment is clearly observed. The critical velocity was observed between 10.5 and 12.0, which confirms the existence of a critical velocity claimed elsewhere.[5.34]

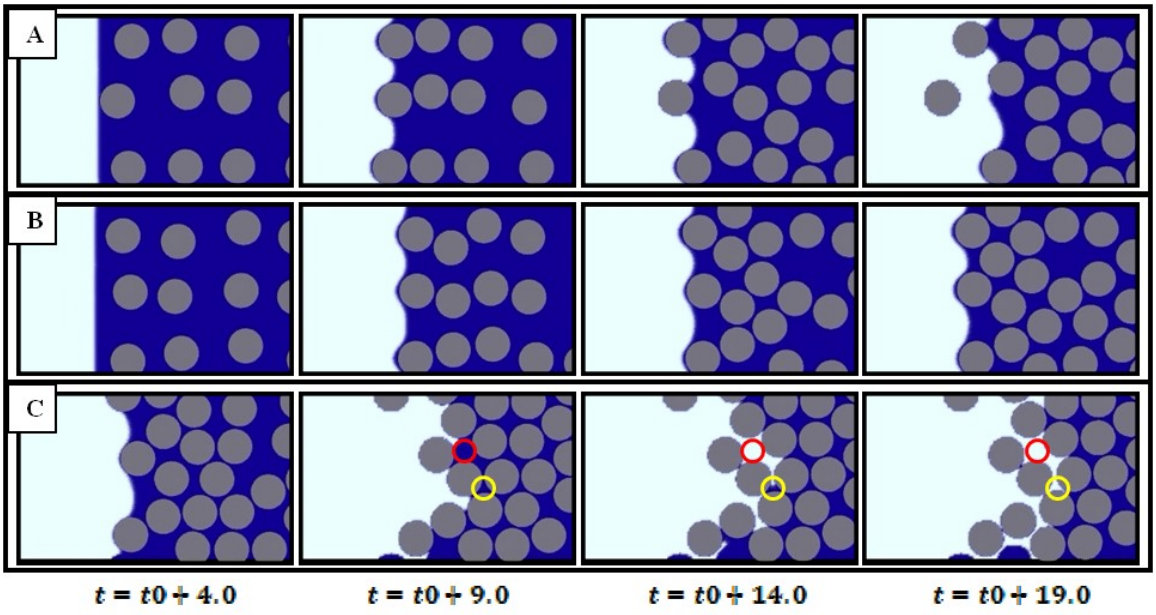


Figure 5.13 Particle-solidification interface dynamics. When the repulsive force between a particle and solid-liquid interface is small (Panel A, k_r = 100, Equation 5.7), particles are engulfed by the moving solid-liquid interface (t = 14.0 and t = 19.0). White zone stands for solid and blue zone is liquid. On the other hand, when the repulsive force between a particle and solid-liquid interface is large (Panel B, k_r = 500), particles are pushed away by the moving interface without being trapped. C) Dynamics showing how solidification entering the pores between near randomly close-packed colloid at a late stage of freezing. The liquid in large pores (red circle) are frozen faster than that in small pores (yellow circle). Note: freezing speed v_s is fixed at 8.0; $t_0 = 0$ for Panel A&B and $t_0 = 30.0$ for Panel C, respectively.

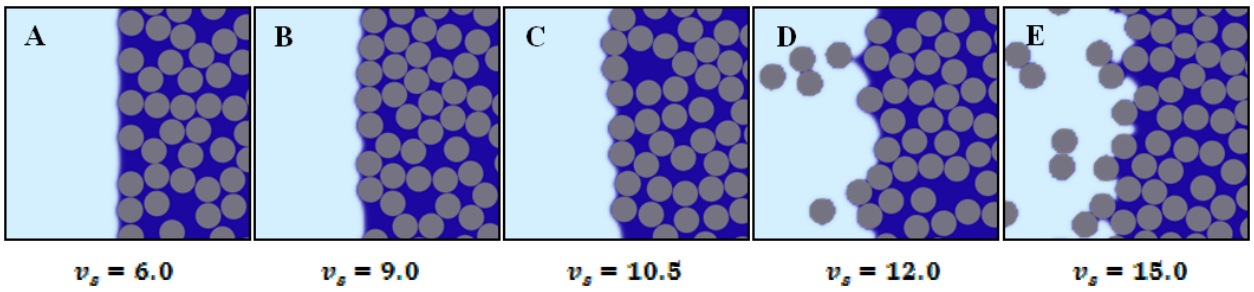


Figure 5.14 Entrapment of particles at different freezing speed v_S: A) 6.0; B) 9.0; C) 10.5; D) 12.0; E) 15.0. A transition from fully repelling to entrapment is clearly observed between v_s = 10.5 and v_s = 12.0, indicating the existence of a critical velocity.

5.4 Summary and future work

To summarize, a fully coupled numerical model was built to simulate the directional freezing of a colloidal suspension in 2D, focusing on the development of self-evolving nonplanar solid-liquid interface, as well as its mutual influence with discrete particle motions. Besides, this model is further modified to explore the detailed particle entrapment at the solid-liquid interface in a finer particle scale. An interactive platform by coupling these two scenarios with a visualization module through high performance GPU devices was also demonstrated. Our results indicate that the instability of the solidification interface is resulted from both thermal and constitutional supercooling. Colloidal particles are more likely rejected without entrapment by evolving solid-liquid interface when a high repelling force between the interface and particles exist or at a low freezing speed. Solid particles are found mostly concentrated between those frozen tips up to a volumetric concentration of about 0.5 in 2D. In this confined environment, freezing solid starts to enter the particle gaps, with liquid harder to freeze in small pores than that in large pores. While our simulation is dimensionless and morphological at this stage, it still shed much light on the solidification based porous material processing. In our future work, the effect

of particle size, the drag force between the interface and the particle, and the attraction force between particles will be further investigated.

References:

[5.1] Y. Tian, P. Mukherjee, T. V. Jayaraman, Z. Xu, Y. Yu, L. Tan, D. J. Sellmyer, and J. E. Shield, *Nano Lett.* **14**, 4328 (2014).

[5.2] M. H. Hoa, P. Y. Kuoa, H. J. Hsieha, T. Y. Hsienb, L. T. Houc, J. Y. Laid, and D. M. Wang, *Biomaterials* **25**, 129 (2004).

[5.3] H. W. Kang, Y. Tabata, and Y. Ikada, *Biomaterials* **20**, 1339 (1999).

[5.4] C. Y. Hsieh, S. P. Tsai, D. M. Wang, Y. N. Chang, and H. J. Hsieh, *Biomaterials* **26**, 5617 (2005).

[5.5] C. Y. Hsieh, S. P. Tsai, M. H. Ho, D. M. Wang, C. E. Liu, C. H. Hsieh, H. C. Tseng, and H. J. Hsieh, *Carbohydr. Polym.* **67**, 124 (2007).

[5.6] W. F. Daamen, H. Th. B. Van Moerkerk, T. Hafmans, L. Buttafoco, A. A. Poot, J. H. Veerkamp, and T. H. van Kuppevelt, *Biomaterials* **24**, 4001 (2003).

[5.7] N. Dagalakis, J. Flink, and P. Stasikelis, *J. Biomed. Mater. Res.* **14**, 511 (1980).

[5.8] W. S. W. Shalaby, G. E. Peck, and K. Park, *J. Controlled Release* **16**, 355 (1991).

[5.9] G. Chen, T. Ushida, and T. Tateishi, *Biomaterials* **22**, 2563 (2001).

[5.10] S. Deville, E. Saiz, R. K. Nalla, and A. P. Tomsia, *Science* **311**, 515 (2006).

[5.11] B.-H. Yoon, C.-S. Park, H.-E. Kim, and Y. H. Koh, *Mater. Lett.* **62**, 1700 (2008).

[5.12] Y. H. Koh, J. J. Sun, and H. E. Kim, *Mater. Lett.* **61**, 1283 (2007).

[5.13] T. Fukasawa, Z. Y. Deng, M. Ando, T. Ohji, and S. Kanzaki, *J. Am. Ceram. Soc.* **85**, 2151 (2002).

[5.14] S. H. Lee, S. H. Jun, H. E. Kim, and Y. H. Koh, *J. Am. Ceram. Soc.* **90**, 2807 (2007).

[5.15] S. Ding, Y. P. Zeng, and D. Jiang, *J. Am. Ceram. Soc.* **90**, 2276 (2007).

[5.16] Z.-Y. Deng, H. R. Fernandes, J. M. Ventura, S. Kannan, and J. M. F. Ferreira, *J. Am. Ceram. Soc.* **90**, 1265 (2007).

[5.17] H. J. Hwang, D. Y. Kim, and J. W. Moon, *Key Eng. Mater.* **510–511**, 906 (2006).

[5.18] J. Tang, Y. Chen, H. Wang, H. Liu, and Q. Fan, *Key Eng. Mater.* **280–283**, 1287 (2005).

[5.19] T. Fukasawa, M. Ando, T. Ohji, and S. Kanzaki, *J. Am. Ceram. Soc.* **84**, 230 (2001).

[5.20] J. Yan, Z. Chen, J. Jiang, L. Tan, and X. C. Zeng, *Adv. Mater.* **21**, 314 (2009).

[5.21] S. Deville, E. Maire, A. Lasalle, A. Bogner, C. Gauthier, J. Leloup, and C. Guizard, J. Am. Ceram. Soc. **92**, 2489 (2009).

[5.22] S. Deville, E. Maire, A. Lasalle, A. Bogner, C. Gauthier, J. Leloup, and C. Guizard, J. Am. Ceram. Soc. **92**, 2497 (2009).

[5.23] S. S. L. Peppin, J. A. W. Elliott, and M. G. Worster, *J. Fluid Mech.* 554, 147 (2006).

[5.24] J. W. Garvin and H. S. Udaykumar, *J. Cryst. Growth* **252**, 451 (2003).

[5.25] J. W. Garvin and H. S. Udaykumar, *J. Cryst. Growth* **252**, 467 (2003).

[5.26] S. A. Barr and E. Luijten, *Acta Mater.* **58**, 709 (2010).

[5.27] Y. Suzuki, G. Sazaki, K. Hashimoto, T. Fujiwara, and Y. Furukawa, J. *Cryst. Growth* **383**, 67 (2013).

[5.28] R. Kobayashi, *Physica D* **63**, 410 (1993).

[5.29] A. Criscione, D. Kintea, Z. Tukovic, S. Jakirlic, I.V. Roisman, C. Tropea, *Int. J. Heat Mass Transfer* **66**, 830 (2013).

[5.30] B. K. Mishra, *Int. J. Miner. Process.* **71**, 73 (2003).

[5.31] S. Green, "Particle Simulation using CUDA", NVIDIA Corporation.

[5.32] Https://www.opengl.org/

[5.33] J. G. Berryman, *Phys. Rev. A* 27 (1983) 1053.

[5.34] D. R. Uhlmann and B. Chalmers, *J. Appl. Phys.* **35**, 2986 (1964).

[5.35] W. W. Mullins and R. F. Sekerka, J. Appl. Phys. **35**, 444 (1964).

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

The pore size in porous materials can range from a few angstroms to several micrometers, leading to broad applications in our daily activities as well as advanced technologies. Through my PhD studies, porous materials at different pore size regions are engineered to meet the design requirements and their novel applications are explored.

MOFs are crystalline microporous materials with pore size ranging from several angstroms to a couple of nanometers. While most studies utilized inner pores by regulating their interactions with small molecules, we chemically engineered Cu-MOFs to bind large molecules on the outer surfaces. The engineered hairy MOFs by a touch of salt with Cu-MOFs have demonstrated 120 times more enhanced binding of GFPs, as well as the capability of stepwise binding. Therefore, our H-MOFs can serve as a microreactor for various biomolecular reactions through this consecutive and selective binding.

In addition, we demonstrated that the building blocks released from the MOFs can be further patched back through a healing process at ambient and low temperature conditions. The solid membranes possess greater mechanical resistance after healing, which can potentially be molded into freestanding 3D objects or mend a broken elastic membrane. The healing processes can be greatly enhanced by patching crystalline gaps with small molecules even at a temperature as low as -56 ºC owning to the sufficient mobility of the assisting reagents. These healing phenomena have the potential applications in MOFs serving as catalysts or hydrogen storage/separation materials, by keeping their structure integrity through multiple cycles of packing or extensive uses.

Noncrystalline ultra-flyweight macroporous aerogels (density 0.5 ~ 5 mg cm^{-3}, pore size ~ 120 nm) were fabricated via freeze-drying followed by a chemical reduction process. In these all carbon aerogel, giant graphene sheets are interconnected while CNTs serve as ribs. These UFAs indicates good performance in energy delay of shock wave in our Split-Hopkinson bar experiments, owing to a large amount of air trapped in the extremely high porous and hydrophobic UFAs. More robust structured materials were also attempted by loading graphene sheets into stable porous media such as glass microfiber filters, and higher energy delay was observed.

In order to study the effect of multiple processing parameters on the formation of porous materials, a numerical model was built to simulate the growth dynamics in directional freezing of colloidal suspensions. By coupling the continuum phase-field model of solute freezing with discrete element model of colloidal particles, the mutual influence of the evolving non-planar solidification interface and a large number of colloidal particles ahead of the solidification front were simulated for the first time. The interactive simulation was achieved by programming both simulation module and visualization module on high performance GPU devices. Our results indicate that the instability of the solidification interface can be resulted from both normal and constitutional supercooling. The colloidal particles can be either rejected or engulfed by the moving interface depending on the moving speed of the solid-liquid interface and particle-interface interactions.

However, it has to be pointed out that the simulation work is preliminary and not yet a quantitative prediction. The quantitative prediction on experimental length and time scale is a major challenge due to the fact that the phase-field simulations are simply not feasible if real physical parameters are used. The simulations can only be performed with diffusion length and

dissipation time scale orders of magnitude larger than in a real material, even using the most advanced supercomputer in the world. We will not attempt this limit in our future work. However, the mapping between simulation parameters and real experiment quantities is still very important to make the simulation predictions useful to guide the experiments. In the meantime, we will also implement much larger scale simulation on parallel GPU clusters, which greatly reduces the numerical error in resolving the collision and particle concentration calculation.

Significant energy delay has been observed in our graphene-absorbed glass microfiber filter. However, graphene sheets are found to be hard to enter inner layers of glass microfiber due to the limitation of the pore size. Rigid porous metal thin films with micrometer pores are also fabricated and under investigation. In the meantime, a temperature gauge can be installed to track the temperature fluctuation in the sample cup during Kolsky bar compression test. In order to fully understand the mechanism of air bubbles in energy delay of shock wave, quantitative measurements of trapped air (e.g. size, shape, connectivity) are required.

APPENDICES

A1: Derivation of Equations in Chapter 5

The original derivation of most of the equations can be found in Kobayashi's work.[A-1] In order to get Equation 5.1, consider the following Ginzburg-Landau type free energy with m as a parameter:

$$\emptyset[p;m] = \int \left\{\frac{1}{2}\varepsilon^2|\nabla p|^2 + F(p;m)\right\} d\vec{r}, \tag{A-1}$$

where $p(\vec{r},t)$ is an ordering parameter at the position $\vec{r}$ and time t, and ε is a small parameter determining the thickness of the layer. ε is a microscopic interaction length and also controls the mobility of the interface. F is a double-well potential with local minimums at $p = 0$ and 1 for any given m satisfying $|m| < \frac{1}{2}$, which corresponding to liquid and solid state, respectively:

$$F(p;m) = \frac{1}{4}p^4 - \left(\frac{1}{2} - \frac{1}{3}m\right)p^3 + \left(\frac{1}{4} - \frac{1}{2}m\right)p^2. \tag{A-2}$$

The difference of the two local minimum values is proportional to m, which gives the difference of chemical potentials between liquid and solid states.

Anisotropic is introduced by assuming that ε only depends on the direction of the outer normal vector $\vec{v} = -\nabla p$ at the interface.

$$\emptyset[p;m] = \int \left\{\frac{1}{2}\varepsilon(-\nabla p)^2|\nabla p|^2 + F(p;m)\right\} d\vec{r}, \tag{A-3}$$

From the formula $\tau\frac{\partial p}{\partial t} = -\frac{\delta\emptyset}{\delta p}$, the following evolution equation can be obtained:

$$\tau\frac{\partial p}{\partial t} = -\nabla\cdot\left(|\nabla p|^2\varepsilon\frac{\partial\varepsilon}{\partial\vec{v}}\right) + \nabla\cdot(\varepsilon^2\nabla p) + p(1-p)(p-\frac{1}{2}+m), \tag{A-4}$$

where τ is a small positive constant and m determines the thermodynamical driving force. In 2-D, $\varepsilon = \varepsilon(\theta)$ can be taken where θ is an angle between $\vec{v}$ and a certain direction (usually the positive direction of the x-axis), which leads to Equation 5.1.

In order to get Equation 5.3, letting $\varepsilon = \bar{\varepsilon}\sigma(\theta)$, Equation 5.1 can be rewritten as

$$\tau \frac{\partial p}{\partial t} = \bar{\varepsilon}^2 \left\{ -\frac{\partial}{\partial x}\left(\frac{1}{2}\sigma^2\right)' \frac{\partial p}{\partial y} + \frac{\partial}{\partial y}\left(\frac{1}{2}\sigma^2\right)' \frac{\partial p}{\partial x} + \nabla \cdot (\sigma^2 \nabla p) \right\} + p(1-p)(p - \tfrac{1}{2} + m), \quad \text{(A-5)}$$

where the prime means $d/d\theta$. Assuming the axi-symmetric solution p and using polar coodinates, the terms in the bracket {} can be converted as:

$$-\frac{\partial}{\partial x}\left(\frac{1}{2}\sigma^2\right)' \frac{\partial p}{\partial y} + \frac{\partial}{\partial y}\left(\frac{1}{2}\sigma^2\right)' \frac{\partial p}{\partial x} = \left(\frac{1}{2}\sigma^2\right)'' \frac{1}{r}\frac{\partial p}{\partial r}, \quad \text{(A-6)}$$

$$\nabla \cdot (\sigma^2 \nabla p) = \sigma^2 \left(\frac{\partial^2 p}{\partial r^2} + \frac{1}{r}\frac{\partial p}{\partial r}\right). \quad \text{(A-7)}$$

Stretching the coordinate $\vec{r}$ near the interface by a factor $\bar{\varepsilon}\sigma$ and assuming the traveling front solution in the stretched interval, we have

$$\frac{d^2 p}{dl^2} + \frac{1}{\sigma}\left\{\tau V + \left[\sigma^2 + \left(\frac{1}{2}\sigma^2\right)''\right] k\bar{\varepsilon}\right\}\frac{dp}{dl} + p(1-p)\left(p - \frac{1}{2} + m\right) = 0, \quad \text{(A-8)}$$

where l is a traveling coordinate in the stretched interval. Solving this non-linear eigenvalue problem[A-2], we have Equation 5.3 as a singular limit equation.

Reference:

[A-1] R. Kobayashi, *Physica D* **63**, 410 (1993).

[A-2] P. C. Fife and J. B. Mcleod, *Arch. Rat. Mech. Anal.* **65**, 335 (1977).

A2: CUDA Code for Simulations in Chapter 5

Part I: CUDA code for Section 5.3.1 and 5.3.2.

Usage: 1) install CUDA Toolkit 6.5; 2) save the code to files indicated in the header; 3) Run Makefile.

```
// !!! directionalFreezing.cpp
////////////////////////////////////////////////////////////////////////////
// Main entry for the directional freezing simulation
////////////////////////////////////////////////////////////////////////////

#ifdef _WIN32
#  define WINDOWS_LEAN_AND_MEAN
#  define NOMINMAX
#  include <windows.h>
#endif

// includes
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <GL/glew.h>

#include <cuda_runtime.h>
#include <cuda_gl_interop.h>
#include <curand_kernel.h>

#include <helper_cuda.h>
#include <helper_cuda_gl.h>

#include <helper_functions.h>
#include <math_constants.h>

#if defined(__APPLE__) || defined(MACOSX)
#include <GLUT/glut.h>
#else
#include <GL/freeglut.h>
#endif

#include <rendercheck_gl.h>
#include "paramgl.h"
#include "parameters.h"
#include "tga.h"

const char *sTitle = "Directional Freezing Simulation";

////////////////////////////////////////////////////////////////////////////
// constants
// visualization window size
uint windowW = 1024, windowH = 512;

// phase field grid dimensions
const uint gridWidth = 512;
const uint gridHeight = 128;
const uint numParticles = 30000; // assume = 3 * n^2

const uint numSimulationSteps = 20000;
const uint printFreq = 5000;

// OpenGL vertex buffers
GLuint particlePosVertexBuffer;
GLuint posVertexBuffer;
GLuint phaseVertexBuffer;
GLuint tempVertexBuffer;
GLuint forceVertexBuffer;
GLuint particleDensVertexBuffer; // particle density at underlying phase-field grid points
struct cudaGraphicsResource *cuda_phaseVB_resource, *cuda_tempVB_resource,
*cuda_particleVB_resource;
struct cudaGraphicsResource *cuda_forceVB_resource, *cuda_particleDensVB_resource;

GLuint indexBuffer;
GLuint shaderProg;
GLuint particleShaderProg;
char *vertShaderPath = 0, *fragShaderPath = 0;
char *vertParticleShaderPath = 0, *fragParticleShaderPath = 0;
```

```
// growth cell parameters
float growthCellPos = 0.0;
float T0 =   -1.0; // temperature of cold source
float T1 =   2.0; // temperature of hot source
float slideSpeed = 8.0;
float growthCellLength = 3.0;

// freezing and temperature depression coefficients
float freezeCoeff = 0.4;
float tempDepressCoeff = 1.0;

// phase-field related arrays
float2 *d_dpdr = 0;
float *d_esq = 0;
float *d_ededth = 0;

float *d_particleVolumeMap = 0;
float *d_liquidVolumeMap = 0;

float *h_temp = 0;
float *h_phase = 0;

// particles related arrays
float *h_particleDens = 0;
float *h_particlePos = 0;
float *d_particleVel = 0;
float *d_sortedPos = 0;
float *d_sortedVel = 0;
uint  *d_isParticleFrozen = 0;
//grid data for sorting method
uint  *d_gridParticleHash = 0;
uint  *d_gridParticleIndex = 0;
uint  *d_cellStart = 0;
uint  *d_cellEnd = 0;
//
float phaseFieldSpacing = 0.03;
float particleRadius = 0.005;
float globalDamping = 0.2f;
float gravity = 0.0f;

float visRadius = 3.0 * phaseFieldSpacing;

// DEM model
float collideSpring = 500.0f;
float collideDamping = 0.02f;
float collideShear = 0.1f;
float collideAttraction = 0.0f;
float pThreshold = 0.9;

// cuda RNG
curandState *d_RNGstate;

SimParams h_params;

// pointers to device object
float *g_pptr = NULL;
float *g_tptr = NULL;
// particles
float *g_cptr = NULL;
float *g_fptr = NULL; // force
float *g_dptr = NULL; // particle density

//fps
#define REFRESH_DELAY     1 //ms
static int fpsCount = 0;
static int fpsLimit = 1;
StopWatchInterface *globalTimer = NULL;
StopWatchInterface *fpsTimer = NULL;
uint frameCount = 0;
```

```
// mouse controls
int mouseOldX, mouseOldY;
int mouseButtons = 0;
float rotateX = 90.0f, rotateY = 0.0f;
float translateX = 0.0f, translateY = 0.0f, translateZ = -2.0f;

// keyboard controls
bool animate = true;
bool drawPoints = false;
bool showParticles = true;
bool wireFrame = false;
bool g_hasDouble = false;
int  switchField = 0;

// parameter adjustment slider
ParamListGL *paramsGL;

// export image
GLubyte *image = 0;

////////////////////////////////////////////////////////////////////////////////
// kernels

extern "C"
{
void setParameters(SimParams *hostParams);

void curandInit(curandState *state, uint width, uint height);

void cudaGenerateHelperFieldKernel(float  *d_phaseMap,
                                   float2 *d_dpdrMap,
                                   float  *d_esqMap,
                                   float  *d_ededthMap,
                                   uint    width,
                                   uint    height);

void cudaCalculateParticleVolumeKernel(uint  *isParticleFrozen,
                                       float *d_particlePos,
                                       float *phaseMap,
                                       float *d_particleVolumeMap,
                                       float  radius,
                                       uint   width,
                                       uint   height,
                                       uint   numParticles);

void cudaCalculateLiquidVolumeKernel(float  *d_phaseMap,
                                     float  *d_liquidVolumeMap,
                                     float   radius,
                                     uint    width,
                                     uint    height);

void cudaCalculateParticleDensityKernel(float  *d_particleVolumeMap,
                                        float  *d_liquidVolumeMap,
                                        float  *d_particleDensMap,
                                        uint    width,
                                        uint    height);


void cudaUpdateFieldmapKernel(float   cellPos,
                              float  *d_phaseMap,
                              float  *d_tempMap,
                              float2 *d_dpdrMap,
                              float  *d_esqMap,
                              float  *d_ededthMap,
                              float  *d_particleDensMap,
                              uint    width,
                              uint    height,
                                   curandState *state);

void calcHash(uint  *gridParticleHash,
              uint  *gridParticleIndex,
```

```
              float *pos,
              int    numParticles);

void reorderDataAndFindCellStart(uint  *cellStart,
                                 uint  *cellEnd,
                                 float *sortedPos,
                                 float *sortedVel,
                                 uint  *gridParticleHash,
                                 uint  *gridParticleIndex,
                                 float *oldPos,
                                 float *oldVel,
                                 uint   numParticles,
                                 uint   numCells);

void collide(uint  *isParticleFrozen,
             float *newVel,
             float *forceMap,
             float *sortedPos,
             float *sortedVel,
             float *phaseMap,
             uint  *gridParticleIndex,
             uint  *cellStart,
             uint  *cellEnd,
             uint   numParticles,
             uint   numCells);

void sortParticles(uint *dGridParticleHash, uint *dGridParticleIndex, uint numParticles);

void cudaUpdateParticleKernel(uint  *isParticleFrozen,
                              float *particlePos,
                                   float *particleVel,
                              uint   nParticles,
                              float *phaseMap,
                              uint   width,
                              uint   height);
void cudaUpdateParticleFrozenKernel(uint   *isParticleFrozen,
                                    float  *particlePos,
                                    uint    nParticles,
                                    float  *phaseMap,
                                    uint    width,
                                    uint    height);
} //extern "C"



////////////////////////////////////////////////////////////////////////////////
// forward declarations
void runGraphicsTest(int argc, char **argv);

// GL functionality
bool initGL(int *argc, char **argv);
void createVBO(GLuint *vbo, int size);
void deleteVBO(GLuint *vbo);
void createMeshIndexBuffer(GLuint *id, int w, int h);
void createMeshPositionVBO(GLuint *id, int w, int h);

// generate n particles within w * h space
void createParticlePositionVBO(GLuint *id, int w, int h, int n);
GLuint loadGLSLProgram(const char *vertFileName, const char *fragFileName);

// rendering callbacks
void display();
void keyboard(unsigned char key, int x, int y);
void mouse(int button, int state, int x, int y);
void motion(int x, int y);
void special(int k, int x, int y);
void reshape(int w, int h);
void reset();
void cleanup();
// Cuda functionality
void runCuda();
```

```
// initialization
void generate_field(float *h_p, float *h_t, int w, int h);
void initParticles(float *h_p, int w, int h, int n);
void initHostParams(SimParams *params);
void initParams();
void updateParams(SimParams *params);

// random float generator
float randf() { return (float) rand() / (float) RAND_MAX; }

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv)
{
    srand(time(NULL));
    sdkCreateTimer(&globalTimer);
    sdkCreateTimer(&fpsTimer);
    sdkStartTimer(&globalTimer);
    printf("[%s]\n\n"
           "Left mouse button          - rotate\n"
           "Middle mouse button        - pan\n"
           "Right mouse button         - zoom\n"
           "'w' key                    - toggle wireframe\n"
           "'p' key                    - point mode\n"
           "'s' key                    - switch field\n", sTitle);

    runGraphicsTest(argc, argv);

    cudaDeviceReset();
    exit(EXIT_SUCCESS);
}

////////////////////////////////////////////////////////////////////////////////
//! Run test
////////////////////////////////////////////////////////////////////////////////
void runGraphicsTest(int argc, char **argv)
{
    printf("[%s] ", sTitle);
    printf("\n");

    // First initialize OpenGL context, so we can properly set the GL for CUDA.
    // This is necessary in order to achieve optimal performance with OpenGL/CUDA interop.
    if (false == initGL(&argc, argv))
    {
        cudaDeviceReset();
        return;
    }

    findCudaGLDevice(argc, (const char **)argv);

    initHostParams(&h_params);
    setParameters(&h_params);

    int phaseFieldSize =  gridWidth * gridHeight * sizeof(float);
    int particleArraySize = 4 * numParticles * sizeof(float);

// allocate arrays on host memory
    h_phase = (float *) malloc(phaseFieldSize);
    h_temp = (float *) malloc(phaseFieldSize);

    h_particlePos = (float *) malloc(particleArraySize);
    h_particleDens = (float *) malloc(phaseFieldSize);
// allocate arrays on device memory
    checkCudaErrors(cudaMalloc((void **)&d_dpdr, gridWidth * gridHeight * sizeof(float2)));
    checkCudaErrors(cudaMalloc((void **)&d_esq, phaseFieldSize));
    checkCudaErrors(cudaMalloc((void **)&d_ededth, phaseFieldSize));
    checkCudaErrors(cudaMalloc((void **)&d_RNGstate, gridWidth * gridHeight *
sizeof(curandState)));
    checkCudaErrors(cudaMalloc((void **)&d_particleVolumeMap, phaseFieldSize));
```

```
    checkCudaErrors(cudaMalloc((void **)&d_liquidVolumeMap, phaseFieldSize));

    checkCudaErrors(cudaMalloc((void **)&d_isParticleFrozen, numParticles * sizeof(uint)));
    cudaMemset(d_isParticleFrozen, 0, numParticles * sizeof(uint));
    checkCudaErrors(cudaMalloc((void **)&d_particleVel, particleArraySize));
    cudaMemset(d_particleVel, 0, particleArraySize);
    checkCudaErrors(cudaMalloc((void **)&d_sortedPos, particleArraySize));
    checkCudaErrors(cudaMalloc((void **)&d_sortedVel, particleArraySize));
    checkCudaErrors(cudaMalloc((void **)&d_gridParticleHash, numParticles * sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_gridParticleIndex, numParticles * sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_cellStart, h_params.numParticleGridCells *
sizeof(uint)));
    cudaMemset(d_cellStart, 0xffffffff, h_params.numParticleGridCells * sizeof(uint));
    checkCudaErrors(cudaMalloc((void **)&d_cellEnd, h_params.numParticleGridCells *
sizeof(uint)));

// initialize particles and phase field on host memory
    generate_field(h_phase, h_temp, gridWidth, gridHeight);
    initParticles(h_particlePos, gridWidth, gridHeight, numParticles);
    curandInit(d_RNGstate, gridWidth, gridHeight);

// vertex buffer
    createVBO(&phaseVertexBuffer, phaseFieldSize);
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_phaseVB_resource, phaseVertexBuffer,
cudaGraphicsMapFlagsWriteDiscard));
    createVBO(&tempVertexBuffer, phaseFieldSize);
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_tempVB_resource, tempVertexBuffer,
cudaGraphicsMapFlagsWriteDiscard));

    createVBO(&particleDensVertexBuffer, phaseFieldSize);
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_particleDensVB_resource,
particleDensVertexBuffer,
                    cudaGraphicsMapFlagsWriteDiscard));

//    createVBO(&particlePosVertexBuffer, 4 * numParticles * sizeof(float));
    createParticlePositionVBO(&particlePosVertexBuffer, gridWidth, gridHeight, numParticles);
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_particleVB_resource,
particlePosVertexBuffer,
                            cudaGraphicsMapFlagsWriteDiscard));

    createVBO(&forceVertexBuffer, 3 * numParticles * sizeof(float));
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_forceVB_resource, forceVertexBuffer,
cudaGraphicsMapFlagsWriteDiscard));

// initialize the array VB resources
    size_t num_bytes;

    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_particleVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_cptr, &num_bytes,
cuda_particleVB_resource));
    checkCudaErrors(cudaMemcpy(g_cptr, h_particlePos, particleArraySize, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_particleVB_resource, 0));


    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_phaseVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_pptr, &num_bytes,
cuda_phaseVB_resource));
    checkCudaErrors(cudaMemcpy(g_pptr, h_phase, phaseFieldSize, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_phaseVB_resource, 0));

    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_tempVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_tptr, &num_bytes,
cuda_tempVB_resource));
    checkCudaErrors(cudaMemcpy(g_tptr, h_temp, phaseFieldSize, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_tempVB_resource, 0));

    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_forceVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_fptr, &num_bytes,
cuda_forceVB_resource));
    checkCudaErrors(cudaMemset(g_fptr, 0, 3 * numParticles * sizeof(float)));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_forceVB_resource, 0));
```

```
    // create vertex and index buffer for mesh
    createMeshPositionVBO(&posVertexBuffer, gridWidth, gridHeight);
    createMeshIndexBuffer(&indexBuffer, gridWidth, gridHeight);

    initParams();
    // register callbacks
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMouseFunc(mouse);
    glutMotionFunc(motion);
    glutReshapeFunc(reshape);
//  glutPostRedisplay();
    // start rendering mainloop
    glutMainLoop();
    cudaDeviceReset();
}

////////////////////////////////////////////////////////////////////////////////
//! Run the Cuda kernels
////////////////////////////////////////////////////////////////////////////////

void runCuda()
{
    static int stp = 0;
//    printf("%d\n", stp);
    size_t num_bytes;

// map resources
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_phaseVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_pptr, &num_bytes,
cuda_phaseVB_resource));
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_tempVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_tptr, &num_bytes,
cuda_tempVB_resource));
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_particleVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_cptr, &num_bytes,
cuda_particleVB_resource));
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_forceVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_fptr, &num_bytes,
cuda_forceVB_resource));
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_particleDensVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_dptr, &num_bytes,
cuda_particleDensVB_resource));

// initialize device memory
    cudaMemset(d_particleVolumeMap, 0, gridWidth * gridHeight * sizeof(float));
    cudaMemset(d_liquidVolumeMap, 0, gridWidth * gridHeight * sizeof(float));

// calculate particle density
    cudaCalculateParticleVolumeKernel(d_isParticleFrozen, g_cptr, g_pptr, d_particleVolumeMap,
visRadius, gridWidth,
                                      gridHeight, numParticles);
    cudaCalculateLiquidVolumeKernel(g_pptr, d_liquidVolumeMap, visRadius, gridWidth, gridHeight);
    cudaDeviceSynchronize();
    cudaCalculateParticleDensityKernel(d_particleVolumeMap, d_liquidVolumeMap, g_dptr, gridWidth,
gridHeight);

// update particle entrapment
    cudaUpdateParticleKernel(d_isParticleFrozen, g_cptr, d_particleVel, numParticles, g_pptr,
gridWidth, gridHeight);
    cudaUpdateParticleFrozenKernel(d_isParticleFrozen, g_cptr, numParticles, g_pptr, gridWidth,
gridHeight);

// update phase field
    cudaGenerateHelperFieldKernel(g_pptr, d_dpdr, d_esq, d_ededth, gridWidth, gridHeight);
    cudaDeviceSynchronize();
    cudaUpdateFieldmapKernel(growthCellPos, g_pptr, g_tptr, d_dpdr, d_esq, d_ededth, g_dptr,
gridWidth, gridHeight, d_RNGstate);

// particle collision
    calcHash(d_gridParticleHash, d_gridParticleIndex, g_cptr, numParticles);
```

```
    cudaDeviceSynchronize();
    sortParticles(d_gridParticleHash, d_gridParticleIndex, numParticles);
    cudaDeviceSynchronize();
    reorderDataAndFindCellStart(d_cellStart, d_cellEnd, d_sortedPos, d_sortedVel,
d_gridParticleHash, d_gridParticleIndex,
                                g_cptr, d_particleVel, numParticles,
h_params.numParticleGridCells);
    cudaDeviceSynchronize();
    collide(d_isParticleFrozen, d_particleVel, g_fptr, d_sortedPos, d_sortedVel, g_pptr,
d_gridParticleIndex, d_cellStart, d_cellEnd,
            numParticles, h_params.numParticleGridCells);


    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_phaseVB_resource, 0));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_tempVB_resource, 0));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_particleVB_resource, 0));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_forceVB_resource, 0));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_particleDensVB_resource, 0));

// update growth cell position (hot/cold block positions)
    growthCellPos += slideSpeed * h_params.dt;

// export particle density information
    if(stp > 0 && stp <= numSimulationSteps && stp % printFreq == 0) {
        char buffer[50];
// save particle density along particular line

        checkCudaErrors(cudaMemcpy(h_particleDens, g_dptr, gridWidth * gridHeight * sizeof(float),
cudaMemcpyDeviceToHost));
        FILE *pFile;
        for(int h = 0; h < gridHeight; h+= 10) {
             snprintf(buffer, 50, "pdens_step%d_h%d.txt", stp, h);
             pFile = fopen(buffer, "w");
             if(pFile == NULL) printf("Error open particle density file\n");
             for(int i = 0; i < gridWidth; i++) {
                 int index = h * gridWidth + i;
                 fprintf(pFile, "%f %f\n", i * h_params.spacing, h_particleDens[index]);
             }
             fclose(pFile);
        }
        snprintf(buffer, 50, "pdens_step%d.txt", stp);
        pFile = fopen(buffer, "w");
        for(int i = 0; i < gridWidth; i++) {
            for(int j =0; j < gridHeight; j++) {
                int index = j * gridWidth + i;
                fprintf(pFile, "%d\t%d\t%f\n", i, j, h_particleDens[index]);
            }
        }
        fclose(pFile);

// save image
         int viewport[4];
         glGetIntegerv( GL_VIEWPORT, viewport );

//        printf("view port %d %d %d %d %d %d\n", viewport[0], viewport[1], viewport[2],
viewport[3]
//                                                    , windowW, windowH);

         bool isChanged = false;

         if (windowW != viewport[2]) {
                  windowW = viewport[2];
                  isChanged = true;
         }

         if (windowH != viewport[3]) {
                  windowH = viewport[3];
                  isChanged = true;
         }
```

```
          snprintf(buffer, 50, "%d.tga", stp);
// export frame

          if (image == NULL || isChanged) {
                  image = (GLubyte *)realloc(image, windowW * windowH * sizeof(GLubyte) * 3);
          }

          if (image == NULL) {
                   fprintf(stderr, "Cannot allocate memory!\n");
                   exit(0);
          }

//         printf("step %d windowW %d windowH %d\n", stp, windowW, windowH);
          glReadPixels(0, 0, windowW, windowH, GL_RGB, GL_UNSIGNED_BYTE, image);
        SaveTga(buffer, image, windowW, windowH);
//         free(image);
    }

// end the simulation
    if(stp == numSimulationSteps) {
        SaveTga("any", image, 0, 0);
          float time = sdkGetTimerValue(&globalTimer);
        sdkStopTimer(&globalTimer);
          printf("%d steps take %12.3f s\n", stp, 0.001 * time);
        animate = false;
    }

    ++stp;
}

void computeFPS()
{
    frameCount++;
    fpsCount++;


    if (fpsCount == fpsLimit)
    {
        char fps[256];
        float ifps = 1.f / (sdkGetAverageTimerValue(&fpsTimer) / 1000.f);
        sprintf(fps, "CUDA directional freezing simulation: grid %d X %d, %d colloids %3.1f fps",
                gridWidth, gridHeight, numParticles, ifps);

        glutSetWindowTitle(fps);
        fpsCount = 0;

        fpsLimit = (int)MAX(ifps, 1.f);
        sdkResetTimer(&fpsTimer);
    }
}

////////////////////////////////////////////////////////////////////////////////
//! Display callback
////////////////////////////////////////////////////////////////////////////////
void display()
{
    sdkStartTimer(&fpsTimer);
    // run CUDA kernel to generate vertex positions
    if (animate)
    {
// real time updating simulation or visualzation parameters at each step
        updateParams(&h_params);
        runCuda();
    }

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // set view matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(translateX, translateY, translateZ);
    glRotatef(rotateX, 1.0, 0.0, 0.0);
```

```
    glRotatef(rotateY, 0.0, 1.0, 0.0);

    float scale = gridWidth;
    if(gridWidth / windowW < gridHeight / windowH)
          scale = gridHeight;

//    translateZ = -2.0;
    scale = 4.0 / h_params.spacing / scale;
    glScalef(scale, 1, scale);

    // render from the vbo

    glBindBuffer(GL_ARRAY_BUFFER, posVertexBuffer);
    glVertexPointer(4, GL_FLOAT, 0, 0);
    glEnableClientState(GL_VERTEX_ARRAY);

// choose displaying properties

    if(switchField == 0) {
        glBindBuffer(GL_ARRAY_BUFFER, phaseVertexBuffer);
    }
    else if(switchField == 1) {
        glBindBuffer(GL_ARRAY_BUFFER, tempVertexBuffer);
    }
    else {
        glBindBuffer(GL_ARRAY_BUFFER, particleDensVertexBuffer);
    }

    glClientActiveTexture(GL_TEXTURE0);
    glTexCoordPointer(1, GL_FLOAT, 0, 0);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);

// render particles and phase field to screen
    glUseProgram(shaderProg);

    GLuint uniChosenField, uniT0, uniT1; // assign temperature range for rendering

    uniT0 = glGetUniformLocation(shaderProg, "T0");
    glUniform1f(uniT0, T0);

    uniT1 = glGetUniformLocation(shaderProg, "T1");
    glUniform1f(uniT1, T1);

    uniChosenField = glGetUniformLocation(shaderProg, "chosenField");
    glUniform1i(uniChosenField, switchField);

    if (drawPoints)
    {
        glDrawArrays(GL_POINTS, 0, gridWidth * gridHeight);
    }
    else
    {
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
        glPolygonMode(GL_FRONT_AND_BACK, wireFrame ? GL_LINE : GL_FILL);
        glDrawElements(GL_TRIANGLE_STRIP, ((gridWidth*2)+2)*(gridHeight-1), GL_UNSIGNED_INT, 0);
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    }

    glDisableClientState(GL_VERTEX_ARRAY);
    glClientActiveTexture(GL_TEXTURE0);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);

    glUseProgram(0);

    if(showParticles) {
// draw particles
    glDisable(GL_DEPTH_TEST);
    glBindBuffer(GL_ARRAY_BUFFER, particlePosVertexBuffer);
    glVertexPointer(4, GL_FLOAT, 0, 0);
```

```
    glEnableClientState(GL_VERTEX_ARRAY);

//    glBindBuffer(GL_ARRAY_BUFFER, forceVertexBuffer);
//    glClientActiveTexture(GL_TEXTURE1);
////    glTexCoordPointer(3, GL_FLOAT, 2, (GLvoid*)(sizeof(float)*2));
//    glTexCoordPointer(3, GL_FLOAT, 0, 0);
//    glEnableClientState(GL_TEXTURE_COORD_ARRAY);

    glUseProgram(particleShaderProg);
    glEnable(GL_POINT_SPRITE_ARB);
    glTexEnvi(GL_POINT_SPRITE_ARB, GL_COORD_REPLACE_ARB, GL_TRUE);
    glEnable(GL_VERTEX_PROGRAM_POINT_SIZE_NV);
    glPointSize(2.0f);
    glColor3f(0.5f, 0.5f, 0.5f);

    glDrawArrays(GL_POINTS, 0, numParticles);

    glDisableClientState(GL_VERTEX_ARRAY);
//    glClientActiveTexture(GL_TEXTURE1);
//    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    glDisable(GL_POINT_SPRITE_ARB);
    }

    glUseProgram(0);


    // display parameter sliders
    glDisable(GL_DEPTH_TEST);
    glBlendFunc(GL_ONE_MINUS_DST_COLOR, GL_ZERO); // invert color
    glEnable(GL_BLEND);
    paramsGL->Render(0, 0);
    glDisable(GL_BLEND);
    glEnable(GL_DEPTH_TEST);


    sdkStopTimer(&fpsTimer);
    glutSwapBuffers();
    glutPostRedisplay();
    glutReportErrors();
    computeFPS();
}

// free the resources and memory
void cleanup()
{
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_phaseVB_resource));
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_tempVB_resource));
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_particleVB_resource));
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_forceVB_resource));
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_particleDensVB_resource));

    deleteVBO(&forceVertexBuffer);
    deleteVBO(&particlePosVertexBuffer);
    deleteVBO(&phaseVertexBuffer);
    deleteVBO(&tempVertexBuffer);
    deleteVBO(&posVertexBuffer);
    deleteVBO(&indexBuffer);
    deleteVBO(&particleDensVertexBuffer);

    checkCudaErrors(cudaFree(d_particleVolumeMap));
    checkCudaErrors(cudaFree(d_liquidVolumeMap));
    checkCudaErrors(cudaFree(d_dpdr));
    checkCudaErrors(cudaFree(d_esq));
    checkCudaErrors(cudaFree(d_ededth));
    checkCudaErrors(cudaFree(d_RNGstate));
    checkCudaErrors(cudaFree(d_isParticleFrozen));
    checkCudaErrors(cudaFree(d_particleVel));
    checkCudaErrors(cudaFree(d_sortedPos));
    checkCudaErrors(cudaFree(d_sortedVel));
    checkCudaErrors(cudaFree(d_gridParticleHash));
    checkCudaErrors(cudaFree(d_gridParticleIndex));
```

```
    checkCudaErrors(cudaFree(d_cellStart));
    checkCudaErrors(cudaFree(d_cellEnd));

    free(h_phase);
    free(h_temp);
    free(h_particlePos);
    free(h_particleDens);
    if(image != NULL) {
          free(image);
    }
}

////////////////////////////////////////////////////////////////////////////////
//! Keyboard events handler
////////////////////////////////////////////////////////////////////////////////
void keyboard(unsigned char key, int /*x*/, int /*y*/)
{
    switch (key)
    {
        case (27) :
            cleanup();
            exit(EXIT_SUCCESS);

        case 'w':
            wireFrame = !wireFrame;
            break;

        case 'p':
            drawPoints = !drawPoints;
            break;

        case ' ':
            animate = !animate;
            break;
        case 'd':
            showParticles = !showParticles;
            break;
        case 's':
            if(switchField == 2) {
                switchField = 0;
            }
            else {
                ++switchField;
            }
//            switchField = !switchField;
            break;
        case 'r':
            reset();
            break;
    }
}

////////////////////////////////////////////////////////////////////////////////
//! Mouse event handlers
////////////////////////////////////////////////////////////////////////////////
void mouse(int button, int state, int x, int y)
{
    if (state == GLUT_DOWN)
    {
        mouseButtons |= 1<<button;
    }
    else if (state == GLUT_UP)
    {
        mouseButtons = 0;
    }

    mouseOldX = x;
    mouseOldY = y;

// slider
    if(paramsGL->Mouse(x, y, button, state))
```

```
    {
        glutPostRedisplay();
        return;
    }

    glutPostRedisplay();
}

void motion(int x, int y)
{
    float dx, dy;
    dx = (float)(x - mouseOldX);
    dy = (float)(y - mouseOldY);

    if(paramsGL->Motion(x, y))
    {
        mouseOldX = x;
        mouseOldY = y;
        glutPostRedisplay();
        return;
    }


    if (mouseButtons == 1)
    {
        rotateX += dy * 0.2f;
        rotateY += dx * 0.2f;
    }
    else if (mouseButtons == 2)
    {
        translateX += dx * 0.01f;
        translateY -= dy * 0.01f;
    }
    else if (mouseButtons == 4)
    {
        translateZ += dy * 0.01f;
    }

    mouseOldX = x;
    mouseOldY = y;
}

void reshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (double) w / (double) h, 0.1, 10.0);
//    gluOrtho2D(-w/2, w/2, -h/2, h/2);
//    gluOrtho2D(-1.2, 1.2, -1.2, 1.2);
    glMatrixMode(GL_MODELVIEW);

    windowW = w;
    windowH = h;
}

void reset()
{
    rotateX = 90.0f;
    rotateY = 0.0f;
    translateX = 0.0f;
    translateY = 0.0f;
    translateZ = -2.0f;
}

////////////////////////////////////////////////////////////////////////////////
//! Initialize GL
////////////////////////////////////////////////////////////////////////////////
bool initGL(int *argc, char **argv)
{
    // Create GL context
```

```
    glutInit(argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(windowW, windowH);
    glutCreateWindow("Ice Growth Simulation");

    vertShaderPath = sdkFindFilePath("phaseField.vert", argv[0]);
    fragShaderPath = sdkFindFilePath("phaseField.frag", argv[0]);

    vertParticleShaderPath = sdkFindFilePath("particle.vert", argv[0]);
    fragParticleShaderPath = sdkFindFilePath("particle.frag", argv[0]);


    if (vertShaderPath == NULL || fragShaderPath == NULL)
    {
        fprintf(stderr, "Error unable to find GLSL vertex and fragment shaders!\n");
        exit(EXIT_FAILURE);
    }

    if (vertParticleShaderPath == NULL || fragParticleShaderPath == NULL)
    {
        fprintf(stderr, "Error unable to find GLSL particle vertex and fragment shaders!\n");
        exit(EXIT_FAILURE);
    }


    // initialize necessary OpenGL extensions
    glewInit();

    if (! glewIsSupported("GL_VERSION_2_0 "
                          ))
    {
        fprintf(stderr, "ERROR: Support for necessary OpenGL extensions missing.");
        fflush(stderr);
        return false;
    }

    if (!glewIsSupported("GL_VERSION_1_5 GL_ARB_vertex_buffer_object GL_ARB_pixel_buffer_object"))
    {
        fprintf(stderr, "Error: failed to get minimal extensions for demo\n");
        fprintf(stderr, "This sample requires:\n");
        fprintf(stderr, "  OpenGL version 1.5\n");
        fprintf(stderr, "  GL_ARB_vertex_buffer_object\n");
        fprintf(stderr, "  GL_ARB_pixel_buffer_object\n");
        cleanup();
        exit(EXIT_FAILURE);
    }

    // default initialization
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glEnable(GL_DEPTH_TEST);

    // load shader
    shaderProg = loadGLSLProgram(vertShaderPath, fragShaderPath);
    particleShaderProg = loadGLSLProgram(vertParticleShaderPath, fragParticleShaderPath);

    SDK_CHECK_ERROR_GL();
    return true;
}

////////////////////////////////////////////////////////////////////////////////
//! Create VBO
////////////////////////////////////////////////////////////////////////////////
void createVBO(GLuint *vbo, int size)
{
    // create buffer object
    glGenBuffers(1, vbo);
    glBindBuffer(GL_ARRAY_BUFFER, *vbo);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    SDK_CHECK_ERROR_GL();
```

```
}

////////////////////////////////////////////////////////////////////////////////
//! Delete VBO
////////////////////////////////////////////////////////////////////////////////
void deleteVBO(GLuint *vbo)
{
    glDeleteBuffers(1, vbo);
    *vbo = 0;
}

// create index buffer for rendering quad mesh
void createMeshIndexBuffer(GLuint *id, int w, int h)
{
    int size = ((w*2)+2)*(h-1)*sizeof(GLuint);

    // create index buffer
    glGenBuffersARB(1, id);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, *id);
    glBufferDataARB(GL_ELEMENT_ARRAY_BUFFER, size, 0, GL_STATIC_DRAW);

    // fill with indices for rendering mesh as triangle strips
    GLuint *indices = (GLuint *) glMapBuffer(GL_ELEMENT_ARRAY_BUFFER, GL_WRITE_ONLY);

    if (!indices)
    {
        return;
    }

    for (int y=0; y<h-1; y++)
    {
        for (int x=0; x<w; x++)
        {
            *indices++ = y*w+x;
            *indices++ = (y+1)*w+x;
        }

        // start new strip with degenerate triangle
        *indices++ = (y+1)*w+(w-1);
        *indices++ = (y+1)*w;
    }

    glUnmapBuffer(GL_ELEMENT_ARRAY_BUFFER);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}

// create vertex buffer to store particle vertices
void createParticlePositionVBO(GLuint *id, int w, int h, int n)
{
    createVBO(id, n * 4 * sizeof(float));
    glBindBuffer(GL_ARRAY_BUFFER, *id);
    float *pos = (float *) glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

    if (!pos)
    {
        return;
    }
// real scale
    for(int i = 0; i < n; i++) {
        *pos++ = (randf() - 0.5f) * w * h_params.spacing;
        *pos++ = 0.0f;
        *pos++ = (randf() - 0.5f) * h * h_params.spacing;
        *pos++ = 1.0f;
    }
    glUnmapBuffer(GL_ARRAY_BUFFER);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

// create fixed vertex buffer to store mesh vertices
void createMeshPositionVBO(GLuint *id, int w, int h)
{
```

```
    createVBO(id, w*h*4*sizeof(float));

    glBindBuffer(GL_ARRAY_BUFFER, *id);
    float *pos = (float *) glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

    if (!pos)
    {
        return;
    }

    for (int y=0; y<h; y++)
    {
        for (int x=0; x<w; x++)
        {
// real scale
             *pos++ = (x - w / 2) * h_params.spacing;
            *pos++ = 0.0f;
            *pos++ = (y - h / 2) * h_params.spacing;
            *pos++ = 1.0f;
        }
    }

    glUnmapBuffer(GL_ARRAY_BUFFER);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

// Attach shader to a program
int attachShader(GLuint prg, GLenum type, const char *name)
{
    GLuint shader;
    FILE *fp;
    int size, compiled;
    char *src;

    fp = fopen(name, "rb");

    if (!fp)
    {
        return 0;
    }

    fseek(fp, 0, SEEK_END);
    size = ftell(fp);
    src = (char *)malloc(size);

    fseek(fp, 0, SEEK_SET);
    fread(src, sizeof(char), size, fp);
    fclose(fp);

    shader = glCreateShader(type);
    glShaderSource(shader, 1, (const char **)&src, (const GLint *)&size);
    glCompileShader(shader);
    glGetShaderiv(shader, GL_COMPILE_STATUS, (GLint *)&compiled);

    if (!compiled)
    {
        char log[2048];
        int len;

        glGetShaderInfoLog(shader, 2048, (GLsizei *)&len, log);
        printf("Info log: %s\n", log);
        glDeleteShader(shader);
        return 0;
    }

    free(src);

    glAttachShader(prg, shader);
    glDeleteShader(shader);

    return 1;
```

```
}

// Create shader program from vertex shader and fragment shader files
GLuint loadGLSLProgram(const char *vertFileName, const char *fragFileName)
{
    GLint linked;
    GLuint program;

    program = glCreateProgram();

    if (!attachShader(program, GL_VERTEX_SHADER, vertFileName))
    {
        glDeleteProgram(program);
        fprintf(stderr, "Couldn't attach vertex shader from file %s\n", vertFileName);
        return 0;
    }

    if (!attachShader(program, GL_FRAGMENT_SHADER, fragFileName))
    {
        glDeleteProgram(program);
        fprintf(stderr, "Couldn't attach fragment shader from file %s\n", fragFileName);
        return 0;
    }

    glLinkProgram(program);
    glGetProgramiv(program, GL_LINK_STATUS, &linked);

    if (!linked)
    {
        glDeleteProgram(program);
        char temp[256];
        glGetProgramInfoLog(program, 256, 0, temp);
        fprintf(stderr, "Failed to link program: %s\n", temp);
        return 0;
    }

    return program;
}


// generate initial phase and temperature field
void generate_field(float *h_p, float *h_t, int w, int h) {
    int idx = 0;

    for(int j = 0; j < h; j++) {
          for(int i = 0; i < w; i++) {
/*
            if(i == 0) {
                h_p[idx] = 1.0;
                h_t[idx] = Ts;
            }
            else {
                h_p[idx] = 0.0;
                h_t[idx] = Tm;
            }
*/
           if(i * phaseFieldSpacing < growthCellLength) {
                h_t[idx] = T0 + i * phaseFieldSpacing / growthCellLength * (T1 - T0);
                if(h_t[idx] <= 0) {
                    h_p[idx] = 1.0;
                }
                else {
                    h_p[idx] = 0.0;
                }
            }
            else {
                h_t[idx] = T1;
                h_p[idx] = 0.0;
            }

            ++idx;
```

```
        }
    }
}

void initHostParams(SimParams *params) {
// grid properties
        params->boundary = make_int4(4, 4, 2, 2); // mirror = 0, Neumann   = 1, periodic  = 2,
parallel  = 3, Dirichlet = 4
        params->gridWidth   = gridWidth;
        params->gridHeight  = gridHeight;
        params->T0 = T0;
        params->T1 = T1;

        params->spacing = phaseFieldSpacing; // spacing
        params->worldOrigin = make_float3(-0.5 * gridWidth * phaseFieldSpacing, 0.0f, -0.5 *
gridHeight * phaseFieldSpacing);

// phase field
        params->alpha   = 0.9;
        params->gamma   = 10.0;
        params->epsb    = 0.01;
        params->tao     = 0.0003;
        params->D       = 1.0;
        params->a       = 0.1;

        params->theta0  = 0.5 * M_PI;
        params->Te      = 1.0;
        params->K       = 1.0;
        params->delta   = 0.05;
        params->j       = 4.0;
// time step
        params->dt      = 0.0001;
// particle
        params->gravity = make_float3(-gravity, 0, 0);
        params->particleRadius  = particleRadius;
        params->globalDamping = globalDamping;
        params->particleGridCellSize = 2.0f * particleRadius; // set the cell size to be the
diameter of the particle
        params->particleGridWidth = (int) ceil(0.5f * gridWidth * phaseFieldSpacing /
particleRadius);
        params->particleGridHeight = (int) ceil(0.5f * gridHeight * phaseFieldSpacing /
particleRadius);
        params->numParticleGridCells = params->particleGridWidth * params->particleGridHeight;

        params->collideSpring = collideSpring;
        params->collideDamping = collideDamping;
        params->collideShear = collideShear;
        params->collideAttraction = collideAttraction;
        params->pThreshold = pThreshold;
// slide speed
        params->slideSpeed = slideSpeed;
        params->growthCellLength = growthCellLength;
// others
        params->f = 1.0;
        params->freezeCoeff = freezeCoeff;
        params->tempDepressCoeff = tempDepressCoeff;
}

void initParticles(float *h_p, int w, int h, int n)
{

    float *pos = h_p;

    // generate uniform distribution of particles
    // suppose n = 3 * k^2
    // volumetric density
    int nh = sqrt(n / 3);
    float s = h * h_params.spacing / nh;
    float initZ = h_params.worldOrigin.z + 0.5 * s;
    // ahead of slider
```

```
    float initX = h_params.worldOrigin.x + growthCellLength + growthCellPos + 0.5 *
h_params.spacing;
    int k = 0;
    while(k < n) {
        int col = k / nh;
        int row = k - col * nh;
        *pos++ = initX + col * s + (randf() - 0.5f) * 0.5 * s;
        *pos++ = 0.0f;
        float tmp = initZ + row * s + (randf() - 0.5f) * 0.5 * s;
        if(tmp < h_params.worldOrigin.z) {
            tmp += h * h_params.spacing;
        }
        else if(tmp >= h_params.worldOrigin.z + h * h_params.spacing) {
            tmp -= h * h_params.spacing;
        }
        *pos++ = tmp;
        *pos++ = 1.0f;
        ++k;
    }
}


void initParams()
{
     paramsGL = new ParamListGL("misc");
     paramsGL->AddParam(new Param<float>("damping coeff", globalDamping, 0.0f, 1.0f, 0.01f,
&globalDamping));
     paramsGL->AddParam(new Param<float>("colloid radius", particleRadius, 0.01f, 0.05f, 0.001f,
&particleRadius));
     paramsGL->AddParam(new Param<float>("hot/cold slide speed", slideSpeed, 0.0f, 32.0f, 8.0f,
&slideSpeed));
     paramsGL->AddParam(new Param<float>("freezeT depress coeff", tempDepressCoeff, 0.0, 2.0f,
0.01f,
                                         &tempDepressCoeff));
     paramsGL->AddParam(new Param<float>("particle freeze coeff", freezeCoeff, 0.0, 1.0f, 0.01f,
&freezeCoeff));
}

void special(int k, int x, int y)
{
    paramsGL->Special(k, x, y);
}


void updateParams(SimParams *params)
{
     params->globalDamping = globalDamping;
     params->particleRadius = particleRadius;
     params->slideSpeed = slideSpeed;
     params->tempDepressCoeff = tempDepressCoeff;
     params->freezeCoeff = freezeCoeff;
     setParameters(params);
}
```

```
// !!! directionalFreezing_kernel.cu
////////////////////////////////////////////////////////////////////////////
// Kernel functions
////////////////////////////////////////////////////////////////////////////
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <GL/glew.h>

#include <cuda_runtime.h>
#include <cuda_gl_interop.h>
#include <curand_kernel.h>

#include <helper_cuda.h>
#include <helper_cuda_gl.h>

#include <helper_functions.h>
#include <math_constants.h>

#include "thrust/device_ptr.h"
#include "thrust/for_each.h"
#include "thrust/iterator/zip_iterator.h"
#include "thrust/sort.h"

#include "phaseField_kernel.cuh"
#include "particles_kernel.cuh"

//Round a / b to nearest higher integer value
int cuda_iDivUp(int a, int b)
{
    return (a + (b - 1)) / b;
}

void computeGridSize(uint n, uint blockSize, uint &numBlocks, uint &numThreads)
{
    numThreads = min(blockSize, n);
    numBlocks = cuda_iDivUp(n, numThreads);
}

// phase-field kernels

__global__ void setup_RNG_kernel(curandState *state, int seed, uint width, uint height)
{
    uint x = blockIdx.x*blockDim.x + threadIdx.x;
    uint y = blockIdx.y*blockDim.y + threadIdx.y;
    uint i = y * width + x;
    if((x < width) && (y < height)) {
          curand_init(seed, i, 0, &state[i]);
    }

}

// wrapper functions
extern "C"
{

void setParameters(SimParams *hostParams)
{
    checkCudaErrors(cudaMemcpyToSymbol(d_params, hostParams, sizeof(SimParams)));
}

void curandInit(curandState *state, uint width, uint height)
{
    dim3 block(16, 16, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    setup_RNG_kernel<<<grid2, block>>>(state, rand(), width, height);

    // check if kernel invocation generated an error
    getLastCudaError("Kernel execution failed: setup_RNG_kernel");
}
```

```
void cudaGenerateHelperFieldKernel(float  *d_phaseMap,
                                   float2 *d_dpdrMap,
                                   float  *d_esqMap,
                                   float  *d_ededthMap,
                                   uint    width,
                                   uint    height)
{
    dim3 block(16, 16, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    generateHelperFieldKernel<<<grid2, block>>>(d_phaseMap, d_dpdrMap, d_esqMap, d_ededthMap,
width, height);
    getLastCudaError("Kernel execution failed: generateHelperFieldKernel");
}


void cudaUpdateFieldmapKernel(float   cellPos,
                              float  *d_phaseMap,
                              float  *d_tempMap,
                              float2 *d_dpdrMap,
                              float  *d_esqMap,
                              float  *d_ededthMap,
                              float  *d_particleDensMap,
                              uint    width,
                              uint    height,
                                   curandState *state)
{
    dim3 block(16, 16, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    updateFieldmapKernel<<<grid2, block>>>(cellPos, d_phaseMap, d_tempMap, d_dpdrMap,
                                           d_esqMap, d_ededthMap, d_particleDensMap, width,
height, state);
    getLastCudaError("Kernel execution failed: updateFieldmapKernel");
}


// subfunctions for particle collision

void calcHash(uint  *gridParticleHash,
              uint  *gridParticleIndex,
              float *pos,
              int    numParticles)
{
    uint numThreads, numBlocks;
    computeGridSize(numParticles, 256, numBlocks, numThreads);

    // execute the kernel
    calcHashD<<< numBlocks, numThreads >>>(gridParticleHash,
                                           gridParticleIndex,
                                           (float4 *) pos,
                                           numParticles);

    // check if kernel invocation generated an error
    getLastCudaError("Kernel execution failed: clacHashD");
}


void reorderDataAndFindCellStart(uint  *cellStart,
                                 uint  *cellEnd,
                                 float *sortedPos,
                                 float *sortedVel,
                                 uint  *gridParticleHash,
                                 uint  *gridParticleIndex,
                                 float *oldPos,
                                 float *oldVel,
                                 uint   numParticles,
                                 uint   numCells)
{
    uint numThreads, numBlocks;
    computeGridSize(numParticles, 256, numBlocks, numThreads);
```

```
    // set all cells to empty
    checkCudaErrors(cudaMemset(cellStart, 0xffffffff, numCells*sizeof(uint)));

    uint smemSize = sizeof(uint)*(numThreads+1);
    reorderDataAndFindCellStartD<<< numBlocks, numThreads, smemSize>>>(
        cellStart,
        cellEnd,
        (float4 *) sortedPos,
        (float4 *) sortedVel,
        gridParticleHash,
        gridParticleIndex,
        (float4 *) oldPos,
        (float4 *) oldVel,
        numParticles);
    getLastCudaError("Kernel execution failed: reorderDataAndFindCellStartD");

}


void collide(uint  *isParticleFrozen,
             float *newVel,
             float *forceMap,
             float *sortedPos,
             float *sortedVel,
             float *phaseMap,
             uint  *gridParticleIndex,
             uint  *cellStart,
             uint  *cellEnd,
             uint   numParticles,
             uint   numCells)
{
    // thread per particle
    uint numThreads, numBlocks;
    computeGridSize(numParticles, 256, numBlocks, numThreads);

    // execute the kernel
    collideD<<< numBlocks, numThreads >>>(isParticleFrozen,
                                          (float4 *)newVel,
                                          (float3 *)forceMap,
                                          (float4 *)sortedPos,
                                          (float4 *)sortedVel,
                                                    phaseMap,
                                          gridParticleIndex,
                                          cellStart,
                                          cellEnd,
                                          numParticles);

    // check if kernel invocation generated an error
    getLastCudaError("Kernel execution failed: collideD");
}



void sortParticles(uint *dGridParticleHash, uint *dGridParticleIndex, uint numParticles)
{
    thrust::sort_by_key(thrust::device_ptr<uint>(dGridParticleHash),
                        thrust::device_ptr<uint>(dGridParticleHash + numParticles),
                        thrust::device_ptr<uint>(dGridParticleIndex));
}


void cudaUpdateParticleKernel(uint  *isParticleFrozen,
                              float *particlePos,
                                   float *particleVel,
                                   uint   numParticles,
                              float *phaseMap,
                              uint   width,
                              uint   height)
{
    uint numThreads, numBlocks;
```

```
    computeGridSize(numParticles, 256, numBlocks, numThreads);
    updateParticleKernel<<<numBlocks, numThreads>>>(isParticleFrozen, (float4 *) particlePos,
(float4 *) particleVel, numParticles,
                                                    phaseMap, width, height);
    getLastCudaError("Kernel execution failed: updateParticleKernel");
}

void cudaUpdateParticleFrozenKernel(uint   *isParticleFrozen,
                                    float  *particlePos,
                                    uint    numParticles,
                                    float  *phaseMap,
                                    uint    width,
                                    uint    height)
{
    uint numThreads, numBlocks;
    computeGridSize(numParticles, 256, numBlocks, numThreads);
    updateParticleFrozenKernel<<<numBlocks, numThreads>>>(isParticleFrozen, (float4 *)
particlePos, numParticles,
                                   phaseMap, width, height);
    getLastCudaError("Kernel execution failed: updateParticleKernel");
}



void cudaCalculateParticleVolumeKernel(uint  *isParticleFrozen,
                                       float *particlePos,
                                       float *phaseMap,
                                       float *particleVolumeMap,
                                       float  radius,
                                       uint   width,
                                       uint   height,
                                       uint   numParticles)
{
    uint numThreads, numBlocks;
    computeGridSize(numParticles, 256, numBlocks, numThreads);
    calculateParticleVolumeKernel<<<numBlocks, numThreads>>>(isParticleFrozen, (float4 *)
particlePos, phaseMap, particleVolumeMap,
                                       radius, width, height, numParticles);
    getLastCudaError("Kernel execution failed: calculateParticleVolumeKernel");
}


void cudaCalculateLiquidVolumeKernel(float  *phaseMap,
                                     float  *liquidVolumeMap,
                                     float  radius,
                                     uint   width,
                                     uint   height)
{
    dim3 block(16, 16, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    calculateLiquidVolumeKernel<<<grid2, block>>>(phaseMap, liquidVolumeMap, radius, width,
height);
    getLastCudaError("Kernel execution failed: calculateLiquidVolumeKernel");

}

void cudaCalculateParticleDensityKernel(float  *particleVolumeMap,
                                        float  *liquidVolumeMap,
                                        float  *particleDensMap,
                                        uint    width,
                                        uint    height)
{
    dim3 block(16, 16, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    calculateParticleDensityKernel<<<grid2, block>>>(particleVolumeMap, liquidVolumeMap,
particleDensMap, width, height);
    getLastCudaError("Kernel execution failed: calculateParticleDensityKernel");

}
} // extern "C"
```

```
// !!! parameters.h
////////////////////////////////////////////////////////////////////////////////
// Declaration of simulation parameters
////////////////////////////////////////////////////////////////////////////////

#ifndef _PARAMETERS_H_
#define _PARAMETERS_H_

#include <vector_types.h>

typedef unsigned int uint;

struct SimParams
{
// phase-field grid properties
        int4     boundary; // mirror = 0, Neumann   = 1, periodic  = 2, parallel  = 3, Dirichlet
= 4
        uint     gridWidth;
        uint     gridHeight;
        float    T0;
        float    T1;
        float    spacing; // spacing
        float3   worldOrigin;
// phase-field
        float    alpha;
        float    gamma;
        float    epsb;
        float    tao;
        float    D;
        float    a;

        float    theta0;
        float    Te;
        float    K;
        float    delta;
        float    j;
// particle and particle grid
        float3   gravity;
        float    globalDamping;
        float    particleRadius;
        float    particleGridCellSize;
        uint     numParticleGridCells;
        uint     particleGridWidth;
        uint     particleGridHeight;

        float    collideSpring;
        float    collideDamping;
        float    collideShear;
        float    collideAttraction;
        float    pThreshold; // for collide with phase point
// time step
        float    dt;
// slide
        float   slideSpeed;
        float   growthCellLength;
// others, adjust the height of the potential well
        float    f;
// freeze criteria coeff
        float    freezeCoeff;
// freezing temperature depression
        float    tempDepressCoeff;

};

#endif
```

```
// !!! particles_kernel.cuh
////////////////////////////////////////////////////////////////////////////////
// device functions for particle simulation
////////////////////////////////////////////////////////////////////////////////

#ifndef PARTICLES_KERNEL_H
#define PARTICLES_KERNEL_H

#include "vector_types.h"
#include <cstdio>
#include <cmath>
#include "helper_math.h"
#include "math_constants.h"
#include "parameters.h"
#include "phaseField_kernel.cuh"

__device__ int3 calcParticleGridPos(float3 p)
{
    int3 gridPos;
    gridPos.x = floor((p.x - d_params.worldOrigin.x) / d_params.particleGridCellSize);
    gridPos.y = 0;
    gridPos.z = floor((p.z - d_params.worldOrigin.z) / d_params.particleGridCellSize);
    return gridPos;
}

// round to nearest phase grid point
__device__ int3 calcPhaseGridPos(float3 p)
{
    int3 gridPos;
    gridPos.x = floor((p.x - d_params.worldOrigin.x) / d_params.spacing);
    gridPos.y = 0;
    gridPos.z = floor((p.z - d_params.worldOrigin.z) / d_params.spacing);
    return gridPos;
}

__device__ uint calcGridHash(int3 gridPos)
{
    return gridPos.z * d_params.particleGridWidth + gridPos.x;
}

__global__ void calcHashD(uint    *gridParticleHash,
                          uint    *gridParticleIndex,
                          float4  *pos,
                          uint     numParticles)
{
    uint index = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    if(index >= numParticles) return;
    volatile float4 p = pos[index];
    int3 gridPos = calcParticleGridPos(make_float3(p));
    uint hash = calcGridHash(gridPos);

    gridParticleHash[index] = hash;
    gridParticleIndex[index] = index;
}


__global__
void reorderDataAndFindCellStartD(uint   *cellStart,        // output: cell start index
                                  uint   *cellEnd,          // output: cell end index
                                  float4 *sortedPos,        // output: sorted positions
                                  float4 *sortedVel,        // output: sorted velocities
                                  uint   *gridParticleHash, // input: sorted grid hashes
                                  uint   *gridParticleIndex,// input: sorted particle indices
                                  float4 *oldPos,           // input: sorted position array
                                  float4 *oldVel,           // input: sorted velocity array
                                  uint    numParticles)
{
    extern __shared__ uint sharedHash[];    // blockSize + 1 elements
    uint index = __umul24(blockIdx.x,blockDim.x) + threadIdx.x;

    uint hash;
```

```
    // handle case when no. of particles not multiple of block size
    if (index < numParticles)
    {
        hash = gridParticleHash[index];

        // Load hash data into shared memory so that we can look
        // at neighboring particle's hash value without loading
        // two hash values per thread
        sharedHash[threadIdx.x+1] = hash;

        if (index > 0 && threadIdx.x == 0)
        {
            // first thread in block must load neighbor particle hash
            sharedHash[0] = gridParticleHash[index-1];
        }
    }

    __syncthreads();

    if (index < numParticles)
    {
        // If this particle has a different cell index to the previous
        // particle then it must be the first particle in the cell,
        // so store the index of this particle in the cell.
        // As it isn't the first particle, it must also be the cell end of
        // the previous particle's cell

        if (index == 0 || hash != sharedHash[threadIdx.x])
        {
            cellStart[hash] = index;

            if (index > 0)
                cellEnd[sharedHash[threadIdx.x]] = index;
        }

        if (index == numParticles - 1)
        {
            cellEnd[hash] = index + 1;
        }

        // Now use the sorted index to reorder the pos and vel data
        uint sortedIndex = gridParticleIndex[index];
        float4 pos = oldPos[sortedIndex];
        float4 vel = oldVel[sortedIndex];

        sortedPos[index] = pos;
        sortedVel[index] = vel;
    }


}


// collide two spheres using DEM method
__device__
float3 collideSpheres(float3 posA, float3 posB,
                      float3 velA, float3 velB,
                      float radiusA, float radiusB)
{
    float3 force = make_float3(0.0f);
    // calculate relative position
    float3 relPos = posB - posA;
// apply PBC in z
    float h = d_params.spacing * d_params.gridHeight;
    if(relPos.z > 0.5 * h) {
        relPos.z = relPos.z - h;
    }
    else if(relPos.z < -0.5 * h) {
        relPos.z = relPos.z + h;
    }
```

```
    float dist = length(relPos);
    float collideDist = radiusA + radiusB;

    if (dist < collideDist)
    {
        float3 norm = relPos / dist;

        // relative velocity
        float3 relVel = velB - velA;

        // relative tangential velocity
        float3 tanVel = relVel - (dot(relVel, norm) * norm);

        // spring force
        force = -d_params.collideSpring*(collideDist - dist) * norm;

        // dashpot (damping) force
        force += d_params.collideDamping*relVel;
        // tangential shear force
        force += d_params.collideShear*tanVel;
/*
        // attraction
        force += d_params.collideAttraction*relPos;
*/
    }

    return force;
}


// collide a particle against all other particles in a given cell
__device__
float3 collideParticleCell(int3    gridPos,
                           uint    index,
                                float3  pos,
                                float3  vel,
                                float4 *oldPos,
                                float4 *oldVel,
                                uint   *cellStart,
                                uint   *cellEnd)
{
    uint gridHash = calcGridHash(gridPos);

    // get start of bucket for this cell
    uint startIndex = cellStart[gridHash];

    float3 force = make_float3(0.0f);

    if (startIndex != 0xffffffff)          // cell is not empty
    {
        // iterate over particles in this cell
        uint endIndex = cellEnd[gridHash];

        for (uint j=startIndex; j<endIndex; j++)
        {
            if (j != index)                // check not colliding with self
            {
                float3 pos2 = make_float3(oldPos[j]);
                float3 vel2 = make_float3(oldVel[j]);

                // collide two spheres
                force += collideSpheres(pos, pos2, vel, vel2, d_params.particleRadius,
d_params.particleRadius);
            }
        }
    }

    return force;
}
```

```
__device__
float3 collideParticleGrid(uint    index,
                           float3  pos,
                           float3  vel,
                           float4 *oldPos,
                           float4 *oldVel,
                           uint   *cellStart,
                           uint   *cellEnd)
{
    // get address in grid
    int3 gridPos = calcParticleGridPos(pos);

    // examine neighboring cells
    float3 force = make_float3(0.0f);

    for (int z=-1; z<=1; z++)
    {
        for (int x=-1; x<=1; x++)
        {
            int3 neighborPos = gridPos + make_int3(x, 0, z);
            if(neighborPos.x >= 0 && neighborPos.x < d_params.particleGridWidth) {
                check_boundary(neighborPos.z, 0, d_params.particleGridHeight, d_params.boundary.z,
d_params.boundary.w);
                force += collideParticleCell(neighborPos, index, pos, vel, oldPos, oldVel,
cellStart, cellEnd);
            }
        }
    }
    return force;
}


__device__ float3 collidePhasePoint(float3   particlePos,
                                    float3   particleVel,
                                    float    particleRadius,
                                    float3   phasePos)
{
    // calculate relative position
    float3 relPos = phasePos - particlePos;
// apply PBC in z
    float h = d_params.spacing * d_params.gridHeight;
    if(relPos.z > 0.5 * h) {
        relPos.z = relPos.z - h;
    }
    else if(relPos.z < -0.5 * h) {
        relPos.z = relPos.z + h;
    }

    float dist = length(relPos);
    float collideDist = particleRadius + d_params.spacing;

    float3 force = make_float3(0.0f);

    if (dist < collideDist)
    {
        float3 norm = relPos / dist;

        // relative velocity
        float3 relVel = -particleVel;

        // relative tangential velocity
        float3 tanVel = relVel - (dot(relVel, norm) * norm);

        // spring force
        force = -d_params.collideSpring*(collideDist - dist) * norm;
        // dashpot (damping) force
        force += d_params.collideDamping*relVel;
        // tangential shear force
        force += d_params.collideShear*tanVel;
        // attraction
```

```
        force += d_params.collideAttraction*relPos;
    }

    return force;
}

// collid a particle with all other phase points in a given cell
__device__ float3 collidePhaseCell(float3  particlePos,
                                   float3  particleVel,
                                        float  *phaseMap,
                                   uint    width,
                                   uint    height,
                                   float   pThreshold)
{
          float3 force = make_float3(0.0f);
        float3 phasePos = make_float3(0.0f);
        int3 nearestPhaseGridPos = calcPhaseGridPos(particlePos);
        int cellSize = 3;
        int neighborX, neighborZ;
        for(int i = -cellSize; i <= cellSize; i++) {
              neighborX = nearestPhaseGridPos.x + i;
            if(neighborX < 0 || neighborX >= width) {
                continue;
            }
              for(int j = -cellSize; j <= cellSize; j++) {
                neighborZ = nearestPhaseGridPos.z + j;
                check_boundary(neighborZ, 0, height, d_params.boundary.z, d_params.boundary.w);
                   int idx = neighborZ * width + neighborX;
                if(phaseMap[idx] >= pThreshold) {
                    phasePos = make_float3(neighborX * d_params.spacing + d_params.worldOrigin.x,
0.0f,
                                           neighborZ * d_params.spacing + d_params.worldOrigin.z);
                    force += phaseMap[idx] * collidePhasePoint(particlePos, particleVel,
d_params.particleRadius, phasePos);
                }
              }
        }
        return force;
}


__global__
void collideD(uint   *isParticleFrozen,
              float4 *newVel,               // output: new velocity
              float3 *forceMap,             // output: force map
              float4 *oldPos,               // input: sorted positions
              float4 *oldVel,               // input: sorted velocities
              float  *phaseMap,             // input: phase map
              uint   *gridParticleIndex,    // input: sorted particle indices
              uint   *cellStart,
              uint   *cellEnd,
              uint    numParticles)
{
    uint index = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;

    if (index >= numParticles) return;
    uint originalIndex = gridParticleIndex[index];
    if(isParticleFrozen[originalIndex] == 1) {
        forceMap[originalIndex] = make_float3(0.0f);
        return;
    }

    // read particle data from sorted arrays
    float3 pos = make_float3(oldPos[index]);
    float3 vel = make_float3(oldVel[index]);
    float3 force = make_float3(0.0f);

    // force with phase-field
    force += collidePhaseCell(pos, vel, phaseMap, d_params.gridWidth, d_params.gridHeight,
d_params.pThreshold);
    // force with other particles
```

```
    force += collideParticleGrid(index, pos, vel, oldPos, oldVel, cellStart, cellEnd);

    // write new velocity back to original unsorted location
    newVel[originalIndex] = make_float4(vel + force, 0.0f);
    forceMap[originalIndex] = force;
}


__global__ void updateParticleKernel(uint   *isParticleFrozen,
                                     float4 *particlePos,
                                     float4 *particleVel,
                                     uint    nParticles,
                                     float  *phaseMap,
                                     uint    width,
                                     uint    height)
{
    uint index = __mul24(blockIdx.x, blockDim.x) + threadIdx.x;
    if(index >= nParticles) return;
    if(isParticleFrozen[index] == 1) return;

    float3 vel = make_float3(particleVel[index]);
    float3 pos = make_float3(particlePos[index]);
    vel += d_params.gravity * d_params.dt;
    vel *= d_params.globalDamping;
    pos += vel * d_params.dt;

// boundary
    float xlen = width * d_params.spacing;
    float zlen = height * d_params.spacing;
    if(pos.x >=  d_params.worldOrigin.x + d_params.particleRadius &&
       pos.x < d_params.worldOrigin.x + xlen - d_params.particleRadius) {
        particlePos[index].x = pos.x;
    }
    if(pos.z < d_params.worldOrigin.z) {
        particlePos[index].z = pos.z + zlen;
    }
    else if(pos.z >= d_params.worldOrigin.z + zlen) {
        particlePos[index].z = pos.z - zlen;
    }
    else {
        particlePos[index].z = pos.z;
    }
}


__global__ void updateParticleFrozenKernel(uint   *isParticleFrozen,
                                           float4 *particlePos,
                                           uint    nParticles,
                                           float  *phaseMap,
                                           uint    width,
                                           uint    height)
{
    uint index = __mul24(blockIdx.x, blockDim.x) + threadIdx.x;
    if(index >= nParticles) return;
    if(isParticleFrozen[index] == 1) return;

    int cellSize = 2;
    float frozenDistance = d_params.freezeCoeff * (d_params.particleRadius + d_params.spacing);
    float3 pos = make_float3(particlePos[index]);
    int3 nearestPhaseGridPos = calcPhaseGridPos(pos);
    float dx, dz;
    int neighborX, neighborZ;

    float h = d_params.spacing * d_params.gridHeight;
    for(int i = -cellSize; i <= cellSize; i++) {
        neighborX = nearestPhaseGridPos.x + i;
        if(neighborX >= 0 && neighborX < width) {
            for(int j = -cellSize; j <= cellSize; j++) {
                neighborZ = nearestPhaseGridPos.z + j;
                check_boundary(neighborZ, 0, height, d_params.boundary.z, d_params.boundary.w);
                dx = pos.x - neighborX * d_params.spacing - d_params.worldOrigin.x;
```

```
                dz = pos.z - neighborZ * d_params.spacing - d_params.worldOrigin.z;
// apply PBC in z
                if(dz > 0.5 * h) {
                    dz -= h;
                }
                else if(dz < -0.5 * h) {
                    dz += h;
                }
                uint gridIndex = neighborZ * width + neighborX;
                if (phaseMap[gridIndex] >= d_params.pThreshold && dx * dx + dz * dz <=
frozenDistance * frozenDistance) {
                    isParticleFrozen[index] = 1;
                    return;
                }
            }
        }
    }

}



__global__ void calculateParticleVolumeKernel(uint     *isParticleFrozen,
                                               float4   *particlePos,
                                               float    *phaseMap,
                                               float    *particleVolumeMap,
                                               float     radius,
                                               float     width,
                                               float     height,
                                               float     nParticles)
{
    uint index = __mul24(blockIdx.x, blockDim.x) + threadIdx.x;
    if(index >= nParticles) return;
    if(isParticleFrozen[index] == 1) return;

    int cellSize = ceil(radius / d_params.spacing) + 1;
    float3 pos = make_float3(particlePos[index]);
    int3 nearestPhaseGridPos = calcPhaseGridPos(pos);
    float dx, dz;
    int neighborX, neighborZ;

    for(int i = -cellSize; i <= cellSize; i++) {
        for(int j = -cellSize; j <= cellSize; j++) {
            neighborX = nearestPhaseGridPos.x + i;
            if(neighborX >= 0 && neighborX < width) {
                neighborZ = nearestPhaseGridPos.z + j;
                dx = pos.x - neighborX * d_params.spacing - d_params.worldOrigin.x;
                dz = pos.z - neighborZ * d_params.spacing - d_params.worldOrigin.z;
                check_boundary(neighborZ, 0, height, d_params.boundary.z, d_params.boundary.w);
                if(dx * dx + dz * dz <= radius * radius) {
                    uint gridIndex = neighborZ * width + neighborX;
//                    if(phaseMap[gridIndex] < d_params.pThreshold) {
                      particleVolumeMap[gridIndex] += 1.0;
//                    }
                }
            }
        }
    }
}

#endif // PARTICLES_KERNEL_H
```

```
// !!! phaseField_kernel.cuh
////////////////////////////////////////////////////////////////////////////
// device functions for phase field
////////////////////////////////////////////////////////////////////////////

#ifndef PHASEFIELD_KERNEL_H
#define PHASEFIELD_KERNEL_H

#include "vector_types.h"
#include <cstdio>
#include <cmath>
#include "helper_math.h"
#include "math_constants.h"
#include "parameters.h"

__constant__ SimParams d_params;

__device__ int2 position(int k)
{
        int j = k / d_params.gridWidth;
        int i = k % d_params.gridWidth;
        return make_int2(i, j);
}

__device__ void check_boundary(int& x, int x0, int x1, int b0, int b1)
{
        if(x < x0) {
                if(b0 == 1 || b0 == 4) x = x0;
                else if(b0 == 2) x = x + x1 - x0;
                else if(b0 == 0) x = 2 * x0 - x;
        }
        else if(x >= x1) {
                if(b1 == 1 || b1 == 4) x = x1 - 1;
                else if(b1 == 2) x = x - (x1 - x0);
                else if(b1 == 0) x = 2 * (x1 - 1) - x;
        }
}

__device__ bool isDirichlet(int x, int x0, int x1, int b0, int b1)
{
    if(((x == x0) && (b0 == 4)) || ((x == x1 - 1) && (b1 == 4)))
         return true;
    else
         return false;
}

__device__ float fetch(float *field, int2 pos)
{
        int2 p = pos;
        check_boundary(p.x, 0, d_params.gridWidth, d_params.boundary.x, d_params.boundary.y);
        check_boundary(p.y, 0, d_params.gridHeight, d_params.boundary.z, d_params.boundary.w);
        int k = p.y * d_params.gridWidth + p.x;
        return field[k];
}

// fetch element from multi-field, idx = 0 or 1
__device__ float fetch(float2 *field, int2 pos, int idx)
{
        int2 p = pos;
        check_boundary(p.x, 0, d_params.gridWidth, d_params.boundary.x, d_params.boundary.y);
        check_boundary(p.y, 0, d_params.gridHeight, d_params.boundary.z, d_params.boundary.w);
        int k = p.y * d_params.gridWidth + p.x;
        if(idx == 0) {
                return field[k].x;
        }
        else {
                return field[k].y;
        }
}

__device__ float2 gradient(float *field, int2 pos)
```

```
{
        float2 grad;
        float weight = 1.0 / (2.0 * d_params.spacing);
        grad.x = (fetch(field, make_int2(pos.x + 1, pos.y)) - fetch(field, make_int2(pos.x - 1,
pos.y))) * weight;
        grad.y = (fetch(field, make_int2(pos.x, pos.y + 1)) - fetch(field, make_int2(pos.x, pos.y
- 1))) * weight;
        return grad;
}

__device__ float divergence(float2 *field, int2 pos)
{
        float div = 0;
        float weight = 1.0 / (2.0 * d_params.spacing);
        div += (fetch(field, make_int2(pos.x + 1, pos.y), 0) - fetch(field, make_int2(pos.x - 1,
pos.y), 0)) * weight;
        div += (fetch(field, make_int2(pos.x, pos.y + 1), 1) - fetch(field, make_int2(pos.x,
pos.y - 1), 1)) * weight;
        return div;
}


__device__ float laplacian(float *field, int2 pos)
{
        float lap = 0;
        float weight = 1.0 / (d_params.spacing * d_params.spacing);
        lap = 2.0f *
             (fetch(field, make_int2(pos.x + 1, pos.y)) + fetch(field, make_int2(pos.x - 1,
pos.y))
            + fetch(field, make_int2(pos.x, pos.y + 1)) + fetch(field, make_int2(pos.x, pos.y -
1))
            - 4.0 * fetch(field, pos))
            + 0.5f *
             (fetch(field, make_int2(pos.x + 1, pos.y + 1)) + fetch(field, make_int2(pos.x + 1,
pos.y - 1))
            + fetch(field, make_int2(pos.x - 1, pos.y + 1)) + fetch(field, make_int2(pos.x - 1,
pos.y - 1))
            - 4.0 * fetch(field, pos));
        lap = lap / 3.0f * weight;
        return lap;
}

__device__ float curvature(float *field, int2 pos)
{
        float small = 0.001;
         float curv = 0.0f;
        float weight = 1.0 / d_params.spacing;
        float df1, df2;
        for(int i = 0; i <= 1; i++) {
         df1 = fetch(field, make_int2(pos.x + i, pos.y)) - fetch(field, make_int2(pos.x + i - 1,
pos.y));
                df2 = fetch(field, make_int2(pos.x + i - 1, pos.y + 1)) + fetch(field,
make_int2(pos.x + i, pos.y + 1))
                    - fetch(field, make_int2(pos.x + i - 1, pos.y - 1)) - fetch(field,
make_int2(pos.x + i, pos.y - 1));
                if(abs(df1) > small || abs(df2) > small) {
                   curv += (2 * i - 1) * df1 / sqrt(df1 * df1 + 0.0625 * df2 * df2) * weight;
                }
                df1 = fetch(field, make_int2(pos.x, pos.y + i)) - fetch(field, make_int2(pos.x,
pos.y + i - 1));
                df2 = fetch(field, make_int2(pos.x + 1, pos.y + i - 1)) + fetch(field,
make_int2(pos.x + 1, pos.y + i))
                    - fetch(field, make_int2(pos.x - 1, pos.y + i - 1)) - fetch(field,
make_int2(pos.x - 1, pos.y + i));
                if(abs(df1) > small || abs(df2) > small) {
                        curv += (2 * i - 1) * df1 / sqrt(df1 * df1 + 0.0625 * df2 * df2) * weight;
                }
        }
        return curv;
}
```

```
// generate intermediate arrays
__global__ void generateHelperFieldKernel(float  *d_phaseMap,
                                          float2 *d_dpdrMap,
                                          float  *d_esqMap,
                                          float  *d_ededthMap,
                                          uint width,
                                          uint height)
{
    float small = 0.001;
    uint x = blockIdx.x*blockDim.x + threadIdx.x;
    uint y = blockIdx.y*blockDim.y + threadIdx.y;
    if(x >= width || y >= height) return;
    uint index = y * width + x;
    int2 pos = make_int2(x, y);
    float2 dpdr = gradient(d_phaseMap, pos);
    float theta = 0.0f;
    if(abs(dpdr.x) > small || abs(dpdr.y) > small) {
        float dxdr = -dpdr.x / sqrt(dpdr.x * dpdr.x + dpdr.y * dpdr.y);
        dxdr = clamp(dxdr, -1.0f, 1.0f);
        theta = acos(dxdr);
        if(dpdr.y > 0) {
            theta = -theta;
        }
    }
    float jth = d_params.j * (theta - d_params.theta0);
    float eps = d_params.epsb * (1.0 + d_params.delta * cos(jth));
    d_esqMap[index] = eps * eps;
    d_ededthMap[index] = -eps * d_params.epsb * d_params.j * d_params.delta * sin(jth);
    d_dpdrMap[index] = dpdr;
}

// update field
__global__ void updateFieldmapKernel(float   cellPos,
                                     float  *d_phaseMap,
                                     float  *d_tempMap,
                                     float2 *d_dpdrMap,
                                     float  *d_esqMap,
                                     float  *d_ededthMap,
                                     float  *d_particleDensMap,
                                     uint width,
                                     uint height,
                                     curandState *state)
{

    uint x = blockIdx.x*blockDim.x + threadIdx.x;
    uint y = blockIdx.y*blockDim.y + threadIdx.y;
    if(x >= width || y >= height) return;
    uint index = y * width + x;
    int2 pos = make_int2(x, y);
// check if Dirichlet boundary
    if(isDirichlet(x, 0, width, d_params.boundary.x, d_params.boundary.y) ||
       isDirichlet(y, 0, height, d_params.boundary.z, d_params.boundary.w))
    {
        return;
    }
    else {

        float p = d_phaseMap[index];
        float temp = d_tempMap[index];
        float2 dpdr = d_dpdrMap[index];

        if(d_particleDensMap[index] > 0.9) {
            d_particleDensMap[index] = 0.9;
        }
        float Tm = d_params.Te - d_params.tempDepressCoeff * tan(1.6 * d_particleDensMap[index]);
// if use number of particles
//        float Tm = d_params.Te - d_params.tempDepressCoeff * tan(0.4 *
d_particleDensMap[index]);

//        float Tm = d_params.Te;
```

```
        float m = d_params.alpha / M_PI * atan(d_params.gamma * (Tm - temp));
        float2 lap;
        lap.x = laplacian(d_phaseMap, pos);
        lap.y = laplacian(d_tempMap, pos);
        float2 desqdr = gradient(d_esqMap, pos);
        float2 dededthdr = gradient(d_ededthMap, pos);
        float dpdt = 1.0 / d_params.tao * (-dededthdr.x * dpdr.y + dededthdr.y * dpdr.x
                   + desqdr.x * dpdr.x + desqdr.y * dpdr.y + d_esqMap[index] * lap.x
                   + p * (1.0 - p) * (p - 0.5 + m)
                   + d_params.a * p * (1.0 - p) * (curand_uniform(&state[index]) - 0.5));
        d_phaseMap[index] = p + dpdt * d_params.dt;
/*
        float2 dTdr = gradient(d_tempMap, pos);
// do not apply at the interface
        if(dTdr.x < 0) dTdr.x = 0;
        d_tempMap[index] = temp + d_params.dt * (d_params.D * lap.y + d_params.K * dpdt - dTdr.x
* d_params.slideSpeed);
*/
        float posX = x * d_params.spacing;
        if(posX < cellPos + d_params.growthCellLength) {
            d_tempMap[index] = temp + d_params.dt * (d_params.D * lap.y + d_params.K * dpdt
                             - (d_params.T1 - d_params.T0) / d_params.growthCellLength *
d_params.slideSpeed);
        }
        else {
            d_tempMap[index] = temp + d_params.dt * (d_params.D * lap.y + d_params.K * dpdt);
        }
    }
}


__global__ void calculateLiquidVolumeKernel(float  *phaseMap,
                                            float  *liquidVolumeMap,
                                            float   radius,
                                            uint    width,
                                            uint    height)
{
    uint x = blockIdx.x*blockDim.x + threadIdx.x;
    uint y = blockIdx.y*blockDim.y + threadIdx.y;
    if(x >= width || y >= height) return;
    uint index = y * width + x;
// ignore solid phase
    if(phaseMap[index] >= d_params.pThreshold) {
        liquidVolumeMap[index] = 0.0f;
        return;
    }
// scan the square area
    int neighborX, neighborY;
    int r = ceil(radius / d_params.spacing);
    for(int i = -r - 1; i <= r + 1; i++) {
        neighborX = x + i;
        if(neighborX >= 0 && neighborX < width) {
            for(int j = -r - 1; j <= r + 1; j++) {
                neighborY = y + j;
                float dx = i * d_params.spacing;
                float dy = j * d_params.spacing;
                check_boundary(neighborY, 0, height, d_params.boundary.z, d_params.boundary.w);
                if(dx * dx + dy * dy <= radius * radius && phaseMap[neighborY * width + neighborX]
< d_params.pThreshold) {
                    liquidVolumeMap[index] += 1.0f;
                }
            }
        }
    }
}


__global__ void calculateParticleDensityKernel(float  *particleVolumeMap,
                                               float  *liquidVolumeMap,
                                               float  *particleDensMap,
                                               uint    width,
```

```
                                                uint    height)
{
    uint x = blockIdx.x*blockDim.x + threadIdx.x;
    uint y = blockIdx.y*blockDim.y + threadIdx.y;
    if(x >= width || y >= height) return;
    uint index = y * width + x;
    if(liquidVolumeMap[index] > 0) {
//        particleDensMap[index] = particleVolumeMap[index] / liquidVolumeMap[index]
//                                 * M_PI * d_params.particleRadius * d_params.particleRadius /
d_params.spacing / d_params.spacing;
        particleDensMap[index] = particleVolumeMap[index] * pow(d_params.particleRadius, 2) /
pow(3.0 * d_params.spacing, 2);
    }
    else {
        particleDensMap[index] = 0.0f;
    }

}

#endif //PHASEFIELD_KERNEL_H
```

```
// !!! tga.h
//////////////////////////////////////////////////////////////////////////////
// output frames as tga figure
//////////////////////////////////////////////////////////////////////////////
#ifndef __TGA_H__
#define __TGA_H__

#include <stdio.h>
#include <assert.h>
#include <iostream>

using namespace std;

struct A3D_TGA_HEADER{
        unsigned char nID;
        unsigned char nColorMapType;
        unsigned char nImageType;
        unsigned short nColorMapOrigin;
        unsigned char nColorMapLength;
        unsigned char nColorMapWidth;
        unsigned short nXOrigin;
        unsigned short nYOrigin;
        unsigned short nImageWidth;
        unsigned short nImageHeight;
        unsigned short nFlags;
};

void SaveTga(char *filename, const unsigned char *image,
int image_x, int image_y)
{
     static int32_t tgaimagesize = 0;
     static unsigned char *tgaimage = NULL;

     A3D_TGA_HEADER     tgaHeader;

     int imagesize;

     int16_t  i, j;
     FILE  *fp;

     assert(image);

     /* set the tga header */
     memset(&tgaHeader, 0, sizeof(A3D_TGA_HEADER));
     tgaHeader.nID = 0x0;
     tgaHeader.nColorMapType = 0x0;
     tgaHeader.nImageType = 0x2;
     tgaHeader.nColorMapOrigin = 0x0;
     tgaHeader.nColorMapLength = 0x0;
     tgaHeader.nColorMapWidth = 0x0;
     tgaHeader.nXOrigin = 0x0;
     tgaHeader.nYOrigin = 0x0;
     tgaHeader.nImageWidth = image_x;
     tgaHeader.nImageHeight = image_y;
     tgaHeader.nFlags = 0x18;

     /* fill the image */
     imagesize = image_x * image_y;
// test
//     printf("tgaimage address %x image address %x tgaimage size %d image size %d\n", tgaimage,
image, tgaimagesize, imagesize);
// free the space if the image size is 0
     if(imagesize == 0 && tgaimage != NULL) {
         free(tgaimage);
         return;
     }

     if ((tgaimage == NULL) || (imagesize != tgaimagesize)) {

         if (tgaimage != NULL) {
             free(tgaimage);
```

```
        }

        tgaimage = (unsigned char *)
                calloc(sizeof(unsigned char), imagesize * 3);

        assert(tgaimage);

        tgaimagesize = imagesize;
    }

    memset(tgaimage, 0, sizeof(unsigned char) * imagesize * 3);


    for (i = image_y - 1; i >= 0; i--)
        for (j = 0; j < image_x; j++) {
        int n = i * image_x + j;

        tgaimage[n * 3]     = (unsigned char)(image[n * 3 + 2]);
        tgaimage[n * 3 + 1] = (unsigned char)(image[n * 3 + 1]);
        tgaimage[n * 3 + 2] = (unsigned char)(image[n * 3 + 0]);
    }


        fp = fopen(filename, "w+");
        assert(fp);

        if ((fwrite(&tgaHeader, sizeof(A3D_TGA_HEADER), 1, fp) != 1) ||
             (fwrite(tgaimage, sizeof(unsigned char) * imagesize * 3,
1, fp) != 1)){
            cerr << "Error writing tga file" << endl;
        }

        fclose(fp);
// free and reset, not necessary till the end. How do free the space at the end?
//     free(tgaimage);
//     tgaimage = NULL;
//     imagesize = 0;

    return;
}

void flip(unsigned char *image, int image_x, int image_y, int size) {
          int half_y = image_y / 2;
          for(int i = 0; i < half_y; i++) {
                    for(int j = 0; j < image_x; j++) {
                              for(int k = 0; k < size; k++) {
                                        int idx1 = size * (i * image_x + j) + k;
                                        int idx2 = size * ((image_y - i) * image_x + j) + k;
                                        unsigned char tmp = image[idx1];
                                        image[idx1] = image[idx2];
                                        image[idx2] = tmp;
                              }
                    }
          }
}

#endif // __TGA_H__
```

```
// !!! findcudalib.mk from CUDA Toolkit 6.5

#
#  findcudalib.mk is used to find the locations for CUDA libraries and other
#                 Unix Platforms.  This is supported Mac OS X and Linux.
#
#############################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   ifeq ("$(DISTRO)","')
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$$/d"
2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null
   endif
   ifeq ("$(DISTRO)","')
     # third, we can search in /etc/os-release or /etc/{distro}-release
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" | grep -v
"ID" | grep -v "DISTRIB")
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed 's/DISTRIB_RELEASE=//' | grep
-v "DISTRIB_RELEASE")
   endif
endif

# search at Darwin (unix based info)
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))
ifneq ($(DARWIN),)
   SNOWLEOPARD = $(strip $(findstring 10.6, $(shell egrep "<string>10\.6"
/System/Library/CoreServices/SystemVersion.plist)))
```

```
   LION        = $(strip $(findstring 10.7, $(shell egrep "<string>10\.7"
/System/Library/CoreServices/SystemVersion.plist)))
   MOUNTAIN    = $(strip $(findstring 10.8, $(shell egrep "<string>10\.8"
/System/Library/CoreServices/SystemVersion.plist)))
   MAVERICKS   = $(strip $(findstring 10.9, $(shell egrep "<string>10\.9"
/System/Library/CoreServices/SystemVersion.plist)))
endif

# Common binaries
GCC   ?= g++
CLANG ?= /usr/bin/clang

ifeq ("$(OSUPPER)","LINUX")
     NVCC ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
else
  # for some newer versions of XCode, CLANG is the default compiler, so we need to include this
  ifneq ($(MAVERICKS),)
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(CLANG)
        STDLIB ?= -stdlib=libstdc++
  else
        NVCC   ?= $(CUDA_PATH)/bin/nvcc -ccbin $(GCC)
  endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
          OS_SIZE = 32
          OS_ARCH = i686
endif
ifeq ($(x86_64),1)
          OS_SIZE = 64
          OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
          OS_SIZE = 32
          OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # Each Linux Distribuion has a set of different paths.  This applies especially when using
the Linux RPM/debian packages
    ifeq ("$(DISTRO)","ubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","debian")
        CUDAPATH  ?= /usr/lib/nvidia-current
        CUDALINK  ?= -L/usr/lib/nvidia-current
        DFLT_PATH  = /usr/lib
    endif
    ifeq ("$(DISTRO)","suse")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib64
      else
        CUDAPATH  ?=
        CUDALINK  ?=
        DFLT_PATH  = /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","suse linux")
      ifeq ($(OS_SIZE),64)
        CUDAPATH  ?=
        CUDALINK  ?=
```

```
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","opensuse")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","fedora")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhat")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","red")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
  endif
endif
ifeq ("$(DISTRO)","redhatenterpriseworkstation")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH ?= /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH ?= /usr/lib
  endif
endif
ifeq ("$(DISTRO)","centos")
  ifeq ($(OS_SIZE),64)
    CUDAPATH  ?= /usr/lib64/nvidia
    CUDALINK  ?= -L/usr/lib64/nvidia
    DFLT_PATH  = /usr/lib64
  else
    CUDAPATH  ?=
    CUDALINK  ?=
    DFLT_PATH  = /usr/lib
```

```
      endif
    endif

    ifeq ($(ARMv7),1)
      CUDAPATH := /usr/arm-linux-gnueabihf/lib
      CUDALINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        CUDAPATH += $(TARGET_FS)/usr/lib/nvidia-current
        CUDALINK += -L$(TARGET_FS)/usr/lib/nvidia-current
      endif
    endif

  # Search for Linux distribution path for libcuda.so
  CUDALIB ?= $(shell find $(CUDAPATH) $(DFLT_PATH) -name libcuda.so -print 2>/dev/null)

  ifeq ("$(CUDALIB)",'')
      $(info >>> WARNING - CUDA Driver libcuda.so is not found.  Please check and re-install the
NVIDIA driver. <<<)
      EXEC=@echo "[@]"
  endif
else
  # This would be the Mac OS X path if we had to do anything special
endif
```

```
// !!! findgllib.mk from CUDA Toolkit 6.5
#############################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
#############################################################################
#
#  findgllib.mk is used to find the necessary GL Libraries for specific distributions
#               this is supported on Mac OSX and Linux Platforms
#
#############################################################################

# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "[:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "[:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# Determine OS platform and unix distribution
ifeq ("$(OSLOWER)","linux")
   # first search lsb_release
   DISTRO  = $(shell lsb_release -i -s 2>/dev/null | tr "[:upper:]" "[:lower:]")
   DISTVER = $(shell lsb_release -r -s 2>/dev/null)
   # $(info DISTRO1 = $(DISTRO) $(DISTVER))
   ifeq ($(DISTRO),)
     # second search and parse /etc/issue
     DISTRO = $(shell more /etc/issue | awk '{print $$1}' | sed '1!d' | sed -e "/^$$/d"
2>/dev/null | tr "[:upper:]" "[:lower:]")
     DISTVER= $(shell more /etc/issue | awk '{print $$2}' | sed '1!d' 2>/dev/null)
     # $(info DISTRO2 = $(DISTRO) $(DISTVER))
   endif
   ifeq ($(DISTRO),)
     # third, we can search in /etc/os-release or /etc/{distro}-release
     DISTRO = $(shell awk '/ID/' /etc/*-release | sed 's/ID=//' | grep -v "VERSION" | grep -v
"ID" | grep -v "DISTRIB")
     DISTVER= $(shell awk '/DISTRIB_RELEASE/' /etc/*-release | sed 's/DISTRIB_RELEASE=//' | grep
-v "DISTRIB_RELEASE")
     # $(info DISTRO3 = $(DISTRO) $(DISTVER))
   endif
endif

# Take command line flags that override any of these settings
ifeq ($(i386),1)
          OS_SIZE = 32
```

```
          OS_ARCH = i686
endif
ifeq ($(x86_64),1)
          OS_SIZE = 64
          OS_ARCH = x86_64
endif
ifeq ($(ARMv7),1)
          OS_SIZE = 32
          OS_ARCH = armv7l
endif

ifeq ("$(OSUPPER)","LINUX")
    # $(info) >> findgllib.mk -> LINUX path <<<)
    # Each set of Linux Distros have different paths for where to find their OpenGL libraries
reside
    ifeq ("$(DISTRO)","ubuntu")
        GLPATH    ?= /usr/lib/nvidia-current
        GLLINK    ?= -L/usr/lib/nvidia-current
        DFLT_PATH ?= /usr/lib
    endif
    ifeq ("$(DISTRO)","kubuntu")
        GLPATH    ?= /usr/lib/nvidia-current
        GLLINK    ?= -L/usr/lib/nvidia-current
        DFLT_PATH ?= /usr/lib
    endif
    ifeq ("$(DISTRO)","debian")
        GLPATH    ?= /usr/lib/nvidia-current
        GLLINK    ?= -L/usr/lib/nvidia-current
        DFLT_PATH ?= /usr/lib
    endif
    ifeq ("$(DISTRO)","suse")
      ifeq ($(OS_SIZE),64)
        GLPATH    ?= /usr/X11R6/lib64 /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib64 -L/usr/X11R6/lib
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?= /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","suse linux")
      ifeq ($(OS_SIZE),64)
        GLPATH    ?= /usr/X11R6/lib64 /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib64 -L/usr/X11R6/lib
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?= /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","opensuse")
      ifeq ($(OS_SIZE),64)
        GLPATH    ?= /usr/X11R6/lib64 /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib64 -L/usr/X11R6/lib
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?= /usr/X11R6/lib
        GLLINK    ?= -L/usr/X11R6/lib
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","fedora")
      ifeq ($(OS_SIZE),64)
        GLPATH    ?= /usr/lib64/nvidia
        GLLINK    ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?=
        GLLINK    ?=
```

```
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","redhat")
      ifeq ($(OS_SIZE),64)
        GLPATH    ?= /usr/lib64/nvidia
        GLLINK    ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?=
        GLLINK    ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","red")
      ifeq ($(OS_SIZE),64)
        GLPATH    ?= /usr/lib64/nvidia
        GLLINK    ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?=
        GLLINK    ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","redhatenterpriseworkstation")
      ifeq ($(OS_SIZE),64)
        GLPATH    ?= /usr/lib64/nvidia
        GLLINK    ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?=
        GLLINK    ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif
    ifeq ("$(DISTRO)","centos")
      ifeq ($(OS_SIZE),64)
        GLPATH    ?= /usr/lib64/nvidia
        GLLINK    ?= -L/usr/lib64/nvidia
        DFLT_PATH ?= /usr/lib64
      else
        GLPATH    ?=
        GLLINK    ?=
        DFLT_PATH ?= /usr/lib
      endif
    endif


    ifeq ($(ARMv7),1)
      GLPATH := /usr/arm-linux-gnueabihf/lib
      GLLINK := -L/usr/arm-linux-gnueabihf/lib
      ifneq ($(TARGET_FS),)
        GLPATH += $(TARGET_FS)/usr/lib/nvidia-current $(TARGET_FS)/usr/lib/arm-linux-gnueabihf
        GLLINK += -L$(TARGET_FS)/usr/lib/nvidia-current -L$(TARGET_FS)/usr/lib/arm-linux-
gnueabihf
      endif
    endif

  # find libGL, libGLU, libXi,
  GLLIB  := $(shell find $(GLPATH) $(DFLT_PATH) -name libGL.so  -print 2>/dev/null)
  GLULIB := $(shell find $(GLPATH) $(DFLT_PATH) -name libGLU.so -print 2>/dev/null)
  X11LIB := $(shell find $(GLPATH) $(DFLT_PATH) -name libX11.so -print 2>/dev/null)
  XILIB  := $(shell find $(GLPATH) $(DFLT_PATH) -name libXi.so  -print 2>/dev/null)
  XMULIB := $(shell find $(GLPATH) $(DFLT_PATH) -name libXmu.so -print 2>/dev/null)

  ifeq ("$(GLLIB)",'')
      $(info >>> WARNING - libGL.so not found, refer to CUDA Samples release notes for how to
find and install them. <<<)
      EXEC=@echo "[@]"
  endif
  ifeq ("$(GLULIB)",'')
```

```
      $(info >>> WARNING - libGLU.so not found, refer to CUDA Samples release notes for how to
find and install them. <<<)
      EXEC=@echo "[@]"
  endif
  ifeq ("$(X11LIB)",'')
      $(info >>> WARNING - libX11.so not found, refer to CUDA Samples release notes for how to
find and install them. <<<)
      EXEC=@echo "[@]"
  endif
  ifeq ("$(XILIB)",'')
      $(info >>> WARNING - libXi.so not found, refer to CUDA Samples release notes for how to
find and install them. <<<)
      EXEC=@echo "[@]"
  endif
  ifeq ("$(XMULIB)",'')
      $(info >>> WARNING - libXmu.so not found, refer to CUDA Samples release notes for how to
find and install them. <<<)
      EXEC=@echo "[@]"
  endif
else
    # This would be the Mac OS X path if we had to do anything special
endif
```

```
// Makefile, modified from Makefile in CUDA Toolkit 6.5
################################################################################
#
# Copyright 1993-2013 NVIDIA Corporation.  All rights reserved.
#
# NOTICE TO USER:
#
# This source code is subject to NVIDIA ownership rights under U.S. and
# international Copyright laws.
#
# NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THIS SOURCE
# CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT EXPRESS OR
# IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL WARRANTIES WITH
# REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED WARRANTIES OF
# MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
# IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL,
# OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
# OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
# OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
# OR PERFORMANCE OF THIS SOURCE CODE.
#
# U.S. Government End Users.  This source code is a "commercial item" as
# that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting  of
# "commercial computer software" and "commercial computer software
# documentation" as such terms are used in 48 C.F.R. 12.212 (SEPT 1995)
# and is provided to the U.S. Government only as a commercial end item.
# Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1 through
# 227.7202-4 (JUNE 1995), all U.S. Government End Users acquire the
# source code with only those rights set forth herein.
#
################################################################################
#
# Makefile project only supported on Mac OS X and Linux Platforms)
#
################################################################################

include ./findcudalib.mk

# Location of the CUDA Toolkit
CUDA_PATH       ?= /usr/local/cuda-6.5

# internal flags
NVCCFLAGS   := -m${OS_SIZE}
CCFLAGS     :=
NVCCLDFLAGS :=
LDFLAGS     :=

# Extra user flags
EXTRA_NVCCFLAGS   ?=
EXTRA_NVCCLDFLAGS ?=
EXTRA_LDFLAGS     ?=
EXTRA_CCFLAGS     ?=

# OS-specific build flags
ifneq ($(DARWIN),)
  LDFLAGS += -rpath $(CUDA_PATH)/lib
  CCFLAGS += -arch $(OS_ARCH) $(STDLIB)
else
  ifeq ($(OS_ARCH),armv7l)
    ifeq ($(abi),gnueabi)
      CCFLAGS += -mfloat-abi=softfp
    else
      # default to gnueabihf
      override abi := gnueabihf
      LDFLAGS += --dynamic-linker=/lib/ld-linux-armhf.so.3
      CCFLAGS += -mfloat-abi=hard
    endif
  endif
endif

ifeq ($(ARMv7),1)
```

```
NVCCFLAGS += -target-cpu-arch ARM
ifneq ($(TARGET_FS),)
CCFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += --sysroot=$(TARGET_FS)
LDFLAGS += -rpath-link=$(TARGET_FS)/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib
LDFLAGS += -rpath-link=$(TARGET_FS)/usr/lib/arm-linux-$(abi)
endif
endif

# Debug build flags
ifeq ($(dbg),1)
      NVCCFLAGS += -g -G
      TARGET := debug
else
      TARGET := release
endif

ALL_CCFLAGS :=
ALL_CCFLAGS += $(NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(CCFLAGS))
ALL_CCFLAGS += $(EXTRA_NVCCFLAGS)
ALL_CCFLAGS += $(addprefix -Xcompiler ,$(EXTRA_CCFLAGS))

ALL_LDFLAGS :=
ALL_LDFLAGS += $(ALL_CCFLAGS)
ALL_LDFLAGS += $(NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(LDFLAGS))
ALL_LDFLAGS += $(EXTRA_NVCCLDFLAGS)
ALL_LDFLAGS += $(addprefix -Xlinker ,$(EXTRA_LDFLAGS))

# Common includes and paths for CUDA
INCLUDES  := -I../../common/inc
LIBRARIES :=

################################################################################

# Makefile include to help find GL Libraries
EXEC            ?=
include ./findgllib.mk

# OpenGL specific libraries
ifneq ($(DARWIN),)
  # Mac OSX specific libraries and paths to include
  LIBRARIES += -L/System/Library/Frameworks/OpenGL.framework/Libraries
  LIBRARIES += -lGL -lGLU ../../common/lib/darwin/libGLEW.a
  ALL_LDFLAGS += -Xlinker -framework -Xlinker GLUT
else
  LIBRARIES += -L../../common/lib/$(OSLOWER)/$(OS_ARCH) $(GLLINK)
  LIBRARIES += -lGL -lGLU -lX11 -lXi -lXmu -lglut -lGLEW
endif

# CUDA code generation flags
#ifneq ($(OS_ARCH),armv7l)
#GENCODE_SM10    := -gencode arch=compute_10,code=sm_10
#endif
GENCODE_SM20    := -gencode arch=compute_20,code=sm_20
GENCODE_SM30    := -gencode arch=compute_30,code=sm_30 -gencode
arch=compute_35,code=\"sm_35,compute_35\"
GENCODE_FLAGS   := $(GENCODE_SM10) $(GENCODE_SM20) $(GENCODE_SM30)

LIBRARIES += -lcufft

################################################################################

# Target rules
all: build

build: directionalFreezing

directionalFreezing_kernel.o: directionalFreezing_kernel.cu
```

```
	$(EXEC) $(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

directionalFreezing.o: directionalFreezing.cpp
	$(EXEC) $(NVCC) $(INCLUDES) $(ALL_CCFLAGS) $(GENCODE_FLAGS) -o $@ -c $<

directionalFreezing: directionalFreezing.o directionalFreezing_kernel.o
	$(EXEC) $(NVCC) $(ALL_LDFLAGS) -o $@ $+ $(LIBRARIES)
	$(EXEC) mkdir -p ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))
	$(EXEC) cp $@ ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if $(abi),/$(abi))

run: build
	$(EXEC) ./directionalFreezing

clean:
	$(EXEC) rm -f directionalFreezing directionalFreezing.o directionalFreezing_kernel.o
	$(EXEC) rm -rf ../../bin/$(OS_ARCH)/$(OSLOWER)/$(TARGET)$(if
$(abi),/$(abi))/directionalFreezing

clobber: clean
```

```
// !!! ./data/particle.frag
//////////////////////////////////////////////////////////////////////////////
// GLSL fragment shader for particles
//////////////////////////////////////////////////////////////////////////////

varying vec3 force;
varying vec4 pos;

void main()
{
    float PI = 3.1415927;
    float fl = length(force);
    float scale = 2.0 / PI * atan(fl);
    vec4 colorBeg = vec4(0.5, 0.5, 0.5, 0);
    vec4 colorEnd = vec4(1.0, 1.0, 1.0, 0);

    if(scale < 0.0) {
        gl_FragColor = colorBeg;
    }
    else if(scale > 1.0) {
        gl_FragColor = colorEnd;
    }
    else {
        gl_FragColor = (1.0 - scale) * colorBeg + scale * colorEnd;
    }
}
```

```
// !!! ./data/particle.vert
//////////////////////////////////////////////////////////////////////////////
// GLSL vertex shader for particles
//////////////////////////////////////////////////////////////////////////////

varying vec3 force;
varying vec4 pos;

void main()
{
    force      = gl_MultiTexCoord1.xyz;
    pos              = vec4(gl_Vertex.x, 0.0, gl_Vertex.z, 1.0);
    gl_Position      = gl_ModelViewProjectionMatrix * pos;
}
```

```
// !!! ./data/phaseField.frag
//////////////////////////////////////////////////////////////////////////////
// GLSL fragment shader for phase field
//////////////////////////////////////////////////////////////////////////////


varying float field;

uniform int chosenField;
uniform float T0;
uniform float T1;

void main()
{
      vec4 colorBeg, colorEnd;
      if(chosenField == 1) { // temperature
          field = clamp((field - T0) / (T1 - T0), 0.0, 1.0);
          colorBeg = vec4(0, 0, 1, 0);
          colorEnd = vec4(1, 0, 0, 0);
      }
      else if(chosenField == 2) { // particle density
          field = clamp(field, 0.0, 1.0);
          field = sqrt(field);
          colorBeg = vec4(1, 1, 1, 0);
          colorEnd = vec4(1, 0, 0, 0);
      }
      else { // phase
          field = clamp(field, 0.0, 1.0);
          colorBeg = vec4(0.1098, 0.02745, 0.62745, 0);
          colorEnd = vec4(0.8314, 0.9412, 1, 0);
      }

      gl_FragColor = (1.0 - field) * colorBeg + field * colorEnd;
}
```

```
// !!! ./data/phaseField.vert
//////////////////////////////////////////////////////////////////////////////
// GLSL vertex shader for phase field
//////////////////////////////////////////////////////////////////////////////

varying float field;

void main()
{
    field               = gl_MultiTexCoord0.x;
    vec4 pos          = vec4(gl_Vertex.x, 0, gl_Vertex.z, 1.0);
    gl_Position       = gl_ModelViewProjectionMatrix * pos;
}
```

Part II: CUDA code for Section 5.3.3.

Usage: 1) replace corresponding files in Part I; 2) run Makefile.

```
// !!! directionalFreezing.cpp
////////////////////////////////////////////////////////////////////////////
// Main entry for the directional freezing simulation
////////////////////////////////////////////////////////////////////////////

#ifdef _WIN32
#  define WINDOWS_LEAN_AND_MEAN
#  define NOMINMAX
#  include <windows.h>
#endif

// includes
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <GL/glew.h>

#include <cuda_runtime.h>
#include <cuda_gl_interop.h>
#include <curand_kernel.h>

#include <helper_cuda.h>
#include <helper_cuda_gl.h>

#include <helper_functions.h>
#include <math_constants.h>

#if defined(__APPLE__) || defined(MACOSX)
#include <GLUT/glut.h>
#else
#include <GL/freeglut.h>
#endif

#include <rendercheck_gl.h>
#include "paramgl.h"
#include "parameters.h"
#include "tga.h"

const char *sTitle = "Directional Freezing Simulation";

////////////////////////////////////////////////////////////////////////////
// constants
uint windowW = 1024, windowH = 512;

const uint gridWidth = 512;
const uint gridHeight = 128;
const uint numParticles = 108; // assume = 3 * n^2

const uint numSimulationSteps = 50000;
const uint printFreq = 1000;

// grid
float timeStep = 0.0001;
float phaseFieldSpacing = 0.03;
float particleRadius = 5.0 * phaseFieldSpacing;
float globalDamping = 0.1f;
float particleDrawSize = 24.0;

float visRadius = particleRadius;
// DEM model
float gravity = 50000.0f;
float collideSpring = 100.0f;
float collideDamping = 0.02f;
float collideShear = 0.1f;
float collideAttraction = 0.0f;
float pThreshold = 0.5;

// growth cell parameters
float growthCellPos = 0.0;
```

```
float T0 =  -1.0; // temperature of cold source
float T1 =   2.0; // temperature of hot source
float slideSpeed = gridWidth * phaseFieldSpacing / 60.0 / 600.0 / timeStep;
float growthCellLength = 0.4 * gridWidth * phaseFieldSpacing;

//
float freezeCoeff = 0.0;
float tempDepressCoeff = 0.5;

// OpenGL vertex buffers
GLuint particlePosVertexBuffer;
GLuint posVertexBuffer;
GLuint phaseVertexBuffer;
GLuint tempVertexBuffer;
GLuint forceVertexBuffer;
GLuint particleDensVertexBuffer; // particle density at underlying phase-field grid points
GLuint signedDistVertexBuffer;
struct cudaGraphicsResource *cuda_phaseVB_resource, *cuda_tempVB_resource,
*cuda_particleVB_resource;
struct cudaGraphicsResource *cuda_forceVB_resource, *cuda_particleDensVB_resource,
*cuda_signedDistVB_resource;

GLuint indexBuffer;
GLuint shaderProg;
GLuint particleShaderProg;
char *vertShaderPath = 0, *fragShaderPath = 0;
char *vertParticleShaderPath = 0, *fragParticleShaderPath = 0;

// phase-field related arrays
float2 *d_dpdr = 0;
float *d_esq = 0;
float *d_ededth = 0;

float *d_particleVolumeMap = 0;
float *d_liquidVolumeMap = 0;

float *h_temp = 0;
float *h_phase = 0;

// interface tracking
int  *d_isInterface = 0;
float2 *d_distVec = 0;
float maxDist = 20.0 * phaseFieldSpacing;

// particles related arrays
float *h_particleDens = 0;
float *h_particlePos = 0;
float *d_particleVel = 0;
float *d_sortedPos = 0;
float *d_sortedVel = 0;
uint  *d_isParticleFrozen = 0;
//grid data for sorting method
uint  *d_gridParticleHash = 0;
uint  *d_gridParticleIndex = 0;
uint  *d_cellStart = 0;
uint  *d_cellEnd = 0;

// cuda RNG
curandState *d_RNGstate;

SimParams h_params;

// pointers to device object
float *g_pptr = NULL;
float *g_tptr = NULL;
// particles
float *g_cptr = NULL;
float *g_fptr = NULL; // force
float *g_dptr = NULL; // particle density
float *g_sptr = NULL; // signed distance buffer
//fps
```

```
#define REFRESH_DELAY     1 //ms
// fps
static int fpsCount = 0;
static int fpsLimit = 1;
StopWatchInterface *globalTimer = NULL;
StopWatchInterface *fpsTimer = NULL;
uint frameCount = 0;

// mouse controls
int mouseOldX, mouseOldY;
int mouseButtons = 0;
float rotateX = 90.0f, rotateY = 0.0f;
float translateX = 0.0f, translateY = 0.0f, translateZ = -2.0f;

// keyboard controls
bool animate = true;
bool drawPoints = false;
bool showParticles = true;
bool wireFrame = false;
bool g_hasDouble = false;
int  switchField = 0;

// export image
GLubyte *image = 0;

// parameter slider
ParamListGL *paramsGL;

////////////////////////////////////////////////////////////////////////////////
// kernels

extern "C"
{
void setParameters(SimParams *hostParams);

void curandInit(curandState *state, uint width, uint height);

void cudaGenerateHelperFieldKernel(float  *d_phaseMap,
                                   float2 *d_dpdrMap,
                                   float  *d_esqMap,
                                   float  *d_ededthMap,
                                   uint    width,
                                   uint    height);

void cudaCalculateParticleVolumeKernel(uint  *isParticleFrozen,
                                       float *d_particlePos,
                                       float *phaseMap,
                                       float *d_particleVolumeMap,
                                       float  radius,
                                       uint   width,
                                       uint   height,
                                       uint   numParticles);

void cudaCalculateLiquidVolumeKernel(float  *d_phaseMap,
                                     float  *d_liquidVolumeMap,
                                     float   radius,
                                     uint    width,
                                     uint    height);

void cudaCalculateParticleDensityKernel(float  *d_particleVolumeMap,
                                        float  *d_liquidVolumeMap,
                                        float  *d_particleDensMap,
                                        uint    width,
                                        uint    height);


void cudaUpdateFieldmapKernel(float   cellPos,
                              float  *d_phaseMap,
                              float  *d_tempMap,
                              float2 *d_dpdrMap,
                              float  *d_esqMap,
```

```
                                float  *d_ededthMap,
                                float  *d_particleDensMap,
                                uint    width,
                                uint    height,
                                     curandState *state);

void calcHash(uint  *gridParticleHash,
              uint  *gridParticleIndex,
              float *pos,
              int    numParticles);

void reorderDataAndFindCellStart(uint  *cellStart,
                                 uint  *cellEnd,
                                 float *sortedPos,
                                 float *sortedVel,
                                 uint  *gridParticleHash,
                                 uint  *gridParticleIndex,
                                 float *oldPos,
                                 float *oldVel,
                                 uint   numParticles,
                                 uint   numCells);

void collide(uint  *isParticleFrozen,
             float *newVel,
             float *forceMap,
             float *sortedPos,
             float *sortedVel,
             float2 *distVec,
             uint  *gridParticleIndex,
             uint  *cellStart,
             uint  *cellEnd,
             uint   numParticles,
             uint   numCells);

void sortParticles(uint *dGridParticleHash, uint *dGridParticleIndex, uint numParticles);

void cudaUpdateParticleKernel(uint  *isParticleFrozen,
                              float *particlePos,
                                   float *particleVel,
                              uint   nParticles,
                              float *phaseMap,
                              uint   width,
                              uint   height);
void cudaUpdateParticleFrozenKernel(uint   *isParticleFrozen,
                                    float  *particlePos,
                                    uint    nParticles,
                                    float  *phaseMap,
                                    uint    width,
                                    uint    height);

// interface tracking

void cudaCheckInterfaceKernel(int  *isInterface,
                              float *phaseMap,
                              uint   width,
                              uint   height);

// signed distance for the grid
void cudaCalculateSignedDistanceKernel(float   *signedDist,
                                       float2  *distVec,
                                       int     *isInterface,
                                       uint     width,
                                       uint     height);

} //extern "C"



////////////////////////////////////////////////////////////////////////////////
// forward declarations
void runGraphicsTest(int argc, char **argv);
```

```
// GL functionality
bool initGL(int *argc, char **argv);
void createVBO(GLuint *vbo, int size);
void deleteVBO(GLuint *vbo);
void createMeshIndexBuffer(GLuint *id, int w, int h);
void createMeshPositionVBO(GLuint *id, int w, int h);
// generate n particles within w * h space
void createParticlePositionVBO(GLuint *id, int w, int h, int n);
GLuint loadGLSLProgram(const char *vertFileName, const char *fragFileName);

// rendering callbacks
void display();
void keyboard(unsigned char key, int x, int y);
void mouse(int button, int state, int x, int y);
void motion(int x, int y);
void special(int k, int x, int y);
void reshape(int w, int h);
void reset();
void cleanup();
// Cuda functionality
void runCuda();

//initialization host
void generate_field(float *h_p, float *h_t, int w, int h);
void initParticles(float *h_p, int w, int h, int n);
void initHostParams(SimParams *params);
void updateParams(SimParams *params);
// random float
float randf() { return (float) rand() / (float) RAND_MAX; }

// change params
void initParams();

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char **argv)
{
    srand(time(NULL));
    sdkCreateTimer(&globalTimer);
    sdkCreateTimer(&fpsTimer);
    sdkStartTimer(&globalTimer);
    printf("[%s]\n\n"
           "Left mouse button          - rotate\n"
           "Middle mouse button        - pan\n"
           "Right mouse button         - zoom\n"
           "'w' key                    - toggle wireframe\n"
           "'p' key                    - point mode\n"
           "'s' key                    - switch field\n", sTitle);

    runGraphicsTest(argc, argv);

    cudaDeviceReset();
    exit(EXIT_SUCCESS);
}

////////////////////////////////////////////////////////////////////////////////
//! Run test
////////////////////////////////////////////////////////////////////////////////
void runGraphicsTest(int argc, char **argv)
{
    printf("[%s] ", sTitle);
    printf("\n");

    // First initialize OpenGL context, so we can properly set the GL for CUDA.
    // This is necessary in order to achieve optimal performance with OpenGL/CUDA interop.
    if (false == initGL(&argc, argv))
    {
        cudaDeviceReset();
        return;
```

```
    }

    findCudaGLDevice(argc, (const char **)argv);

    initHostParams(&h_params);
    setParameters(&h_params);

    int phaseFieldSize =  gridWidth * gridHeight * sizeof(float);
    int particleArraySize = 4 * numParticles * sizeof(float);

// host
    h_phase = (float *) malloc(phaseFieldSize);
    h_temp = (float *) malloc(phaseFieldSize);

    h_particlePos = (float *) malloc(particleArraySize);
    h_particleDens = (float *) malloc(phaseFieldSize);
// device

    checkCudaErrors(cudaMalloc((void **)&d_dpdr, gridWidth * gridHeight * sizeof(float2)));
    checkCudaErrors(cudaMalloc((void **)&d_esq, phaseFieldSize));
    checkCudaErrors(cudaMalloc((void **)&d_ededth, phaseFieldSize));
    checkCudaErrors(cudaMalloc((void **)&d_RNGstate, gridWidth * gridHeight *
sizeof(curandState)));
    checkCudaErrors(cudaMalloc((void **)&d_particleVolumeMap, phaseFieldSize));
    checkCudaErrors(cudaMalloc((void **)&d_liquidVolumeMap, phaseFieldSize));

    checkCudaErrors(cudaMalloc((void **)&d_isInterface, gridWidth * gridHeight * sizeof(int)));
    checkCudaErrors(cudaMalloc((void **)&d_distVec, 2 * phaseFieldSize));

    checkCudaErrors(cudaMalloc((void **)&d_isParticleFrozen, numParticles * sizeof(uint)));
    cudaMemset(d_isParticleFrozen, 0, numParticles * sizeof(uint));
    checkCudaErrors(cudaMalloc((void **)&d_particleVel, particleArraySize));
    cudaMemset(d_particleVel, 0, particleArraySize);
    checkCudaErrors(cudaMalloc((void **)&d_sortedPos, particleArraySize));
    checkCudaErrors(cudaMalloc((void **)&d_sortedVel, particleArraySize));
    checkCudaErrors(cudaMalloc((void **)&d_gridParticleHash, numParticles * sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_gridParticleIndex, numParticles * sizeof(uint)));
    checkCudaErrors(cudaMalloc((void **)&d_cellStart, h_params.numParticleGridCells *
sizeof(uint)));
    cudaMemset(d_cellStart, 0xffffffff, h_params.numParticleGridCells * sizeof(uint));
    checkCudaErrors(cudaMalloc((void **)&d_cellEnd, h_params.numParticleGridCells *
sizeof(uint)));

    generate_field(h_phase, h_temp, gridWidth, gridHeight);
    initParticles(h_particlePos, gridWidth, gridHeight, numParticles);
    curandInit(d_RNGstate, gridWidth, gridHeight);

    createVBO(&phaseVertexBuffer, phaseFieldSize);
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_phaseVB_resource, phaseVertexBuffer,
cudaGraphicsMapFlagsWriteDiscard));
    createVBO(&tempVertexBuffer, phaseFieldSize);
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_tempVB_resource, tempVertexBuffer,
cudaGraphicsMapFlagsWriteDiscard));

    createVBO(&particleDensVertexBuffer, phaseFieldSize);
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_particleDensVB_resource,
particleDensVertexBuffer,
                    cudaGraphicsMapFlagsWriteDiscard));

    createVBO(&signedDistVertexBuffer, phaseFieldSize);
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_signedDistVB_resource,
signedDistVertexBuffer,
                    cudaGraphicsMapFlagsWriteDiscard));

    // particles
//    createVBO(&particlePosVertexBuffer, 4 * numParticles * sizeof(float));
    createParticlePositionVBO(&particlePosVertexBuffer, gridWidth, gridHeight, numParticles);
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_particleVB_resource,
particlePosVertexBuffer,
                            cudaGraphicsMapFlagsWriteDiscard));
```

```
    createVBO(&forceVertexBuffer, 3 * numParticles * sizeof(float));
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cuda_forceVB_resource, forceVertexBuffer,
cudaGraphicsMapFlagsWriteDiscard));

// initialize the array VB resources
    size_t num_bytes;

    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_particleVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_cptr, &num_bytes,
cuda_particleVB_resource));
    checkCudaErrors(cudaMemcpy(g_cptr, h_particlePos, particleArraySize, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_particleVB_resource, 0));

    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_phaseVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_pptr, &num_bytes,
cuda_phaseVB_resource));
    checkCudaErrors(cudaMemcpy(g_pptr, h_phase, phaseFieldSize, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_phaseVB_resource, 0));

    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_tempVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_tptr, &num_bytes,
cuda_tempVB_resource));
    checkCudaErrors(cudaMemcpy(g_tptr, h_temp, phaseFieldSize, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_tempVB_resource, 0));

    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_forceVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_fptr, &num_bytes,
cuda_forceVB_resource));
    checkCudaErrors(cudaMemset(g_fptr, 0, 3 * numParticles * sizeof(float)));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_forceVB_resource, 0));

    // create vertex and index buffer for mesh
    createMeshPositionVBO(&posVertexBuffer, gridWidth, gridHeight);
    createMeshIndexBuffer(&indexBuffer, gridWidth, gridHeight);

//    runCuda();

    initParams();
    // register callbacks
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMouseFunc(mouse);
    glutMotionFunc(motion);
    glutReshapeFunc(reshape);
//  glutPostRedisplay();
    // start rendering mainloop
    glutMainLoop();
    cudaDeviceReset();
}

////////////////////////////////////////////////////////////////////////////////
//! Run the Cuda kernels
////////////////////////////////////////////////////////////////////////////////

void runCuda()
{
    static int stp = 0;
//    printf("%d\n", stp);
    size_t num_bytes;

// map resources

    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_phaseVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_pptr, &num_bytes,
cuda_phaseVB_resource));
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_tempVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_tptr, &num_bytes,
cuda_tempVB_resource));
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_particleVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_cptr, &num_bytes,
cuda_particleVB_resource));
```

```
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_forceVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_fptr, &num_bytes,
cuda_forceVB_resource));
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_particleDensVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_dptr, &num_bytes,
cuda_particleDensVB_resource));
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_signedDistVB_resource, 0));
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&g_sptr, &num_bytes,
cuda_signedDistVB_resource));

    cudaMemset(d_particleVolumeMap, 0, gridWidth * gridHeight * sizeof(float));
    cudaMemset(d_liquidVolumeMap, 0, gridWidth * gridHeight * sizeof(float));
    cudaCalculateParticleVolumeKernel(d_isParticleFrozen, g_cptr, g_pptr, d_particleVolumeMap,
visRadius, gridWidth,
                                      gridHeight, numParticles);
    cudaCalculateLiquidVolumeKernel(g_pptr, d_liquidVolumeMap, visRadius, gridWidth, gridHeight);
    cudaDeviceSynchronize();
    cudaCalculateParticleDensityKernel(d_particleVolumeMap, d_liquidVolumeMap, g_dptr, gridWidth,
gridHeight);


    cudaUpdateParticleKernel(d_isParticleFrozen, g_cptr, d_particleVel, numParticles, g_pptr,
gridWidth, gridHeight);
    cudaUpdateParticleFrozenKernel(d_isParticleFrozen, g_cptr, numParticles, g_pptr, gridWidth,
gridHeight);


    cudaGenerateHelperFieldKernel(g_pptr, d_dpdr, d_esq, d_ededth, gridWidth, gridHeight);
    cudaDeviceSynchronize();
    cudaUpdateFieldmapKernel(growthCellPos, g_pptr, g_tptr, d_dpdr, d_esq, d_ededth, g_dptr,
gridWidth, gridHeight, d_RNGstate);
    cudaCheckInterfaceKernel(d_isInterface, g_pptr, gridWidth, gridHeight);
    cudaCalculateSignedDistanceKernel(g_sptr, d_distVec, d_isInterface, gridWidth, gridHeight);

    calcHash(d_gridParticleHash, d_gridParticleIndex, g_cptr, numParticles);
    cudaDeviceSynchronize();
    sortParticles(d_gridParticleHash, d_gridParticleIndex, numParticles);
    cudaDeviceSynchronize();
    reorderDataAndFindCellStart(d_cellStart, d_cellEnd, d_sortedPos, d_sortedVel,
d_gridParticleHash, d_gridParticleIndex,
                                g_cptr, d_particleVel, numParticles,
h_params.numParticleGridCells);
    cudaDeviceSynchronize();
    collide(d_isParticleFrozen, d_particleVel, g_fptr, d_sortedPos, d_sortedVel, d_distVec,
d_gridParticleIndex,
            d_cellStart, d_cellEnd, numParticles, h_params.numParticleGridCells);

    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_phaseVB_resource, 0));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_tempVB_resource, 0));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_particleVB_resource, 0));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_forceVB_resource, 0));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_particleDensVB_resource, 0));
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_signedDistVB_resource, 0));

    growthCellPos += slideSpeed * h_params.dt;

    if(stp > 0 && stp <= numSimulationSteps && stp % printFreq == 0) {
        char buffer[50];
// save particle density along particular line
/*
        checkCudaErrors(cudaMemcpy(h_particleDens, g_dptr, gridWidth * gridHeight * sizeof(float),
cudaMemcpyDeviceToHost));
        FILE *pFile;
        for(int h = 0; h < gridHeight; h+= 10) {
             snprintf(buffer, 50, "pdens_step%d_h%d.txt", stp, h);
             pFile = fopen(buffer, "w");
             if(pFile == NULL) printf("Error open particle density file\n");
             for(int i = 0; i < gridWidth; i++) {
                 int index = h * gridWidth + i;
                 fprintf(pFile, "%f %f\n", i * h_params.spacing, h_particleDens[index]);
             }
```

```
            fclose(pFile);
        }
        snprintf(buffer, 50, "pdens_step%d.txt", stp);
        pFile = fopen(buffer, "w");
        for(int i = 0; i < gridWidth; i++) {
            for(int j =0; j < gridHeight; j++) {
                int index = j * gridWidth + i;
                fprintf(pFile, "%d\t%d\t%f\n", i, j, h_particleDens[index]);
            }
        }
        fclose(pFile);
 */
// save image
          int viewport[4];
          glGetIntegerv( GL_VIEWPORT, viewport );

//         printf("view port %d %d %d %d %d %d\n", viewport[0], viewport[1], viewport[2],
viewport[3]
//                                                          , windowW, windowH);

          bool isChanged = false;

          if (windowW != viewport[2]) {
                    windowW = viewport[2];
                    isChanged = true;
          }

          if (windowH != viewport[3]) {
                    windowH = viewport[3];
                    isChanged = true;
          }


          snprintf(buffer, 50, "%d.tga", stp);
// export frame

          if (image == NULL || isChanged) {
                  image = (GLubyte *)realloc(image, windowW * windowH * sizeof(GLubyte) * 3);
          }

          if (image == NULL) {
                    fprintf(stderr, "Cannot allocate memory!\n");
                    exit(0);
          }

//         printf("step %d windowW %d windowH %d\n", stp, windowW, windowH);
          glReadPixels(0, 0, windowW, windowH, GL_RGB, GL_UNSIGNED_BYTE, image);
        SaveTga(buffer, image, windowW, windowH);
//         free(image);
    }

    if(stp == numSimulationSteps) {
        SaveTga("any", image, 0, 0);
          float time = sdkGetTimerValue(&globalTimer);
        sdkStopTimer(&globalTimer);
          printf("%d steps take %12.3f s\n", stp, 0.001 * time);
        animate = false;
    }

    ++stp;
}

void computeFPS()
{
    frameCount++;
    fpsCount++;

    if (fpsCount == fpsLimit)
    {
        char fps[256];
        float ifps = 1.f / (sdkGetAverageTimerValue(&fpsTimer) / 1000.f);
```

```
        sprintf(fps, "CUDA directional freezing simulation: grid %d X %d, %d colloids %3.1f fps",
                gridWidth, gridHeight, numParticles, ifps);

        glutSetWindowTitle(fps);
        fpsCount = 0;

        fpsLimit = (int)MAX(ifps, 1.f);
        sdkResetTimer(&fpsTimer);
    }
}

////////////////////////////////////////////////////////////////////////////////
//! Display callback
////////////////////////////////////////////////////////////////////////////////
void display()
{
    sdkStartTimer(&fpsTimer);
    // run CUDA kernel to generate vertex positions
    if (animate)
    {
        updateParams(&h_params);
        runCuda();
    }

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // set view matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(translateX, translateY, translateZ);
    glRotatef(rotateX, 1.0, 0.0, 0.0);
    glRotatef(rotateY, 0.0, 1.0, 0.0);

    float scale = gridWidth;
    if(gridWidth / windowW < gridHeight / windowH)
         scale = gridHeight;

//    translateZ = -2.0;
    scale = 4.0 / h_params.spacing / scale;
    glScalef(scale, 1, scale);

    // render from the vbo

    glBindBuffer(GL_ARRAY_BUFFER, posVertexBuffer);
    glVertexPointer(4, GL_FLOAT, 0, 0);
    glEnableClientState(GL_VERTEX_ARRAY);

//ice growth

    if(switchField == 0) {
        glBindBuffer(GL_ARRAY_BUFFER, phaseVertexBuffer);
    }
    else if(switchField == 1) {
        glBindBuffer(GL_ARRAY_BUFFER, tempVertexBuffer);
    }
    else if(switchField == 2) {
        glBindBuffer(GL_ARRAY_BUFFER, signedDistVertexBuffer);
    }
    else {
        glBindBuffer(GL_ARRAY_BUFFER, particleDensVertexBuffer);
    }

    glClientActiveTexture(GL_TEXTURE0);
    glTexCoordPointer(1, GL_FLOAT, 0, 0);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);


    glUseProgram(shaderProg);

    GLuint uniChosenField, uniT0, uniT1, uniMaxDist; // assign temperature range for rendering
```

```
    uniT0 = glGetUniformLocation(shaderProg, "T0");
    glUniform1f(uniT0, T0);

    uniT1 = glGetUniformLocation(shaderProg, "T1");
    glUniform1f(uniT1, T1);

    uniMaxDist = glGetUniformLocation(shaderProg, "maxDist");
    glUniform1f(uniMaxDist, maxDist);

    uniChosenField = glGetUniformLocation(shaderProg, "chosenField");
    glUniform1i(uniChosenField, switchField);

    if (drawPoints)
    {
        glPointSize(10.0);
        glDrawArrays(GL_POINTS, 0, gridWidth * gridHeight);
    }
    else
    {
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
        glPolygonMode(GL_FRONT_AND_BACK, wireFrame ? GL_LINE : GL_FILL);
        glDrawElements(GL_TRIANGLE_STRIP, ((gridWidth*2)+2)*(gridHeight-1), GL_UNSIGNED_INT, 0);
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    }

    glDisableClientState(GL_VERTEX_ARRAY);
    glClientActiveTexture(GL_TEXTURE0);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);

    glUseProgram(0);

    if(showParticles) {
// draw particles
    glDisable(GL_DEPTH_TEST);
    glBindBuffer(GL_ARRAY_BUFFER, particlePosVertexBuffer);
    glVertexPointer(4, GL_FLOAT, 0, 0);
    glEnableClientState(GL_VERTEX_ARRAY);


//    glBindBuffer(GL_ARRAY_BUFFER, forceVertexBuffer);
//    glClientActiveTexture(GL_TEXTURE1);
//    glTexCoordPointer(3, GL_FLOAT, 0, 0);
//    glEnableClientState(GL_TEXTURE_COORD_ARRAY);


//    glUseProgram(particleShaderProg);
    glColor4f(0.5f, 0.5f, 0.5f, 0.9);
    glPointSize(particleDrawSize);
    glEnable(GL_POINT_SMOOTH);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    glDrawArrays(GL_POINTS, 0, numParticles);

    glDisableClientState(GL_VERTEX_ARRAY);
//    glClientActiveTexture(GL_TEXTURE1);
//    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    glDisable(GL_POINT_SPRITE_ARB);
    glDisable(GL_POINT_SMOOTH);
    glDisable(GL_BLEND);
    }

//    glUseProgram(0);

    // display sliders
    glDisable(GL_DEPTH_TEST);
    glBlendFunc(GL_ONE_MINUS_DST_COLOR, GL_ZERO); // invert color
    glEnable(GL_BLEND);
    paramsGL->Render(0, 0);
```

```
    glDisable(GL_BLEND);
    glEnable(GL_DEPTH_TEST);


    sdkStopTimer(&fpsTimer);
    glutSwapBuffers();
    glutPostRedisplay();
    glutReportErrors();
    computeFPS();
}

void cleanup()
{
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_phaseVB_resource));
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_tempVB_resource));
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_particleVB_resource));
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_forceVB_resource));
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_particleDensVB_resource));
    checkCudaErrors(cudaGraphicsUnregisterResource(cuda_signedDistVB_resource));

    deleteVBO(&forceVertexBuffer);
    deleteVBO(&particlePosVertexBuffer);
    deleteVBO(&phaseVertexBuffer);
    deleteVBO(&tempVertexBuffer);
    deleteVBO(&posVertexBuffer);
    deleteVBO(&indexBuffer);
    deleteVBO(&particleDensVertexBuffer);
    deleteVBO(&signedDistVertexBuffer);

    checkCudaErrors(cudaFree(d_distVec));
    checkCudaErrors(cudaFree(d_isInterface));
    checkCudaErrors(cudaFree(d_particleVolumeMap));
    checkCudaErrors(cudaFree(d_liquidVolumeMap));
    checkCudaErrors(cudaFree(d_dpdr));
    checkCudaErrors(cudaFree(d_esq));
    checkCudaErrors(cudaFree(d_ededth));
    checkCudaErrors(cudaFree(d_RNGstate));
    checkCudaErrors(cudaFree(d_isParticleFrozen));
    checkCudaErrors(cudaFree(d_particleVel));
    checkCudaErrors(cudaFree(d_sortedPos));
    checkCudaErrors(cudaFree(d_sortedVel));
    checkCudaErrors(cudaFree(d_gridParticleHash));
    checkCudaErrors(cudaFree(d_gridParticleIndex));
    checkCudaErrors(cudaFree(d_cellStart));
    checkCudaErrors(cudaFree(d_cellEnd));

    free(h_phase);
    free(h_temp);
    free(h_particlePos);
    free(h_particleDens);
    if(image != NULL) {
//        free(image);
        SaveTga("any", image, 0, 0);
    }
}

////////////////////////////////////////////////////////////////////////////////
//! Keyboard events handler
////////////////////////////////////////////////////////////////////////////////
void keyboard(unsigned char key, int /*x*/, int /*y*/)
{
    switch (key)
    {
        case (27) :
            cleanup();
            exit(EXIT_SUCCESS);

        case 'w':
            wireFrame = !wireFrame;
            break;
```

```
        case 'p':
            drawPoints = !drawPoints;
            break;

        case ' ':
            animate = !animate;
            break;
        case 'd':
            showParticles = !showParticles;
            break;
        case 's':
            if(switchField == 3) {
                switchField = 0;
            }
            else {
                ++switchField;
            }
//            switchField = !switchField;
            break;
        case 'r':
            reset();
            break;
    }
}

////////////////////////////////////////////////////////////////////////////////
//! Mouse event handlers
////////////////////////////////////////////////////////////////////////////////
void mouse(int button, int state, int x, int y)
{
    if (state == GLUT_DOWN)
    {
        mouseButtons |= 1<<button;
    }
    else if (state == GLUT_UP)
    {
        mouseButtons = 0;
    }

    mouseOldX = x;
    mouseOldY = y;

// slider
    if(paramsGL->Mouse(x, y, button, state))
    {
        glutPostRedisplay();
        return;
    }

    glutPostRedisplay();
}

void motion(int x, int y)
{
    float dx, dy;
    dx = (float)(x - mouseOldX);
    dy = (float)(y - mouseOldY);

    if(paramsGL->Motion(x, y))
    {
        mouseOldX = x;
        mouseOldY = y;
        glutPostRedisplay();
        return;
    }


    if (mouseButtons == 1)
    {
        rotateX += dy * 0.2f;
        rotateY += dx * 0.2f;
```

```
    }
    else if (mouseButtons == 2)
    {
        translateX += dx * 0.01f;
        translateY -= dy * 0.01f;
    }
    else if (mouseButtons == 4)
    {
        translateZ += dy * 0.01f;
    }

    mouseOldX = x;
    mouseOldY = y;
}

void reshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (double) w / (double) h, 0.1, 10.0);
//    gluOrtho2D(-w/2, w/2, -h/2, h/2);
//    gluOrtho2D(-1.2, 1.2, -1.2, 1.2);
    glMatrixMode(GL_MODELVIEW);

    windowW = w;
    windowH = h;
}

void reset()
{
    rotateX = 90.0f;
    rotateY = 0.0f;
    translateX = 0.0f;
    translateY = 0.0f;
    translateZ = -2.0f;
}

////////////////////////////////////////////////////////////////////////////////
//! Initialize GL
////////////////////////////////////////////////////////////////////////////////
bool initGL(int *argc, char **argv)
{
    // Create GL context
    glutInit(argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(windowW, windowH);
    glutCreateWindow("Ice Growth Simulation");

    vertShaderPath = sdkFindFilePath("phaseField.vert", argv[0]);
    fragShaderPath = sdkFindFilePath("phaseField.frag", argv[0]);

    vertParticleShaderPath = sdkFindFilePath("particle.vert", argv[0]);
    fragParticleShaderPath = sdkFindFilePath("particle.frag", argv[0]);


    if (vertShaderPath == NULL || fragShaderPath == NULL)
    {
        fprintf(stderr, "Error unable to find GLSL vertex and fragment shaders!\n");
        exit(EXIT_FAILURE);
    }

    if (vertParticleShaderPath == NULL || fragParticleShaderPath == NULL)
    {
        fprintf(stderr, "Error unable to find GLSL particle vertex and fragment shaders!\n");
        exit(EXIT_FAILURE);
    }


    // initialize necessary OpenGL extensions
    glewInit();
```

```
    if (! glewIsSupported("GL_VERSION_2_0 "
                          ))
    {
        fprintf(stderr, "ERROR: Support for necessary OpenGL extensions missing.");
        fflush(stderr);
        return false;
    }

    if (!glewIsSupported("GL_VERSION_1_5 GL_ARB_vertex_buffer_object GL_ARB_pixel_buffer_object"))
    {
        fprintf(stderr, "Error: failed to get minimal extensions for demo\n");
        fprintf(stderr, "This sample requires:\n");
        fprintf(stderr, "  OpenGL version 1.5\n");
        fprintf(stderr, "  GL_ARB_vertex_buffer_object\n");
        fprintf(stderr, "  GL_ARB_pixel_buffer_object\n");
        cleanup();
        exit(EXIT_FAILURE);
    }

    // default initialization
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glEnable(GL_DEPTH_TEST);

    // load shader
    shaderProg = loadGLSLProgram(vertShaderPath, fragShaderPath);
    particleShaderProg = loadGLSLProgram(vertParticleShaderPath, fragParticleShaderPath);

    SDK_CHECK_ERROR_GL();
    return true;
}

////////////////////////////////////////////////////////////////////////////////
//! Create VBO
////////////////////////////////////////////////////////////////////////////////
void createVBO(GLuint *vbo, int size)
{
    // create buffer object
    glGenBuffers(1, vbo);
    glBindBuffer(GL_ARRAY_BUFFER, *vbo);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    SDK_CHECK_ERROR_GL();
}

////////////////////////////////////////////////////////////////////////////////
//! Delete VBO
////////////////////////////////////////////////////////////////////////////////
void deleteVBO(GLuint *vbo)
{
    glDeleteBuffers(1, vbo);
    *vbo = 0;
}

// create index buffer for rendering quad mesh
void createMeshIndexBuffer(GLuint *id, int w, int h)
{
    int size = ((w*2)+2)*(h-1)*sizeof(GLuint);

    // create index buffer
    glGenBuffersARB(1, id);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, *id);
    glBufferDataARB(GL_ELEMENT_ARRAY_BUFFER, size, 0, GL_STATIC_DRAW);

    // fill with indices for rendering mesh as triangle strips
    GLuint *indices = (GLuint *) glMapBuffer(GL_ELEMENT_ARRAY_BUFFER, GL_WRITE_ONLY);

    if (!indices)
    {
        return;
```

```
    }

    for (int y=0; y<h-1; y++)
    {
        for (int x=0; x<w; x++)
        {
            *indices++ = y*w+x;
            *indices++ = (y+1)*w+x;
        }

        // start new strip with degenerate triangle
        *indices++ = (y+1)*w+(w-1);
        *indices++ = (y+1)*w;
    }

    glUnmapBuffer(GL_ELEMENT_ARRAY_BUFFER);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}

// create vertex buffer to store particle vertices
void createParticlePositionVBO(GLuint *id, int w, int h, int n)
{
    createVBO(id, n * 4 * sizeof(float));
    glBindBuffer(GL_ARRAY_BUFFER, *id);
    float *pos = (float *) glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

    if (!pos)
    {
        return;
    }
// real scale
    for(int i = 0; i < n; i++) {
        *pos++ = (randf() - 0.5f) * w * h_params.spacing;
        *pos++ = 0.0f;
        *pos++ = (randf() - 0.5f) * h * h_params.spacing;
        *pos++ = 1.0f;
    }
    glUnmapBuffer(GL_ARRAY_BUFFER);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

// create fixed vertex buffer to store mesh vertices
void createMeshPositionVBO(GLuint *id, int w, int h)
{
    createVBO(id, w*h*4*sizeof(float));

    glBindBuffer(GL_ARRAY_BUFFER, *id);
    float *pos = (float *) glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

    if (!pos)
    {
        return;
    }

    for (int y=0; y<h; y++)
    {
        for (int x=0; x<w; x++)
        {
// real scale
             *pos++ = (x - w / 2) * h_params.spacing;
            *pos++ = 0.0f;
            *pos++ = (y - h / 2) * h_params.spacing;
            *pos++ = 1.0f;
        }
    }

    glUnmapBuffer(GL_ARRAY_BUFFER);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

// Attach shader to a program
```

```
int attachShader(GLuint prg, GLenum type, const char *name)
{
    GLuint shader;
    FILE *fp;
    int size, compiled;
    char *src;

    fp = fopen(name, "rb");

    if (!fp)
    {
        return 0;
    }

    fseek(fp, 0, SEEK_END);
    size = ftell(fp);
    src = (char *)malloc(size);

    fseek(fp, 0, SEEK_SET);
    fread(src, sizeof(char), size, fp);
    fclose(fp);

    shader = glCreateShader(type);
    glShaderSource(shader, 1, (const char **)&src, (const GLint *)&size);
    glCompileShader(shader);
    glGetShaderiv(shader, GL_COMPILE_STATUS, (GLint *)&compiled);

    if (!compiled)
    {
        char log[2048];
        int len;

        glGetShaderInfoLog(shader, 2048, (GLsizei *)&len, log);
        printf("Info log: %s\n", log);
        glDeleteShader(shader);
        return 0;
    }

    free(src);

    glAttachShader(prg, shader);
    glDeleteShader(shader);

    return 1;
}

// Create shader program from vertex shader and fragment shader files
GLuint loadGLSLProgram(const char *vertFileName, const char *fragFileName)
{
    GLint linked;
    GLuint program;

    program = glCreateProgram();

    if (!attachShader(program, GL_VERTEX_SHADER, vertFileName))
    {
        glDeleteProgram(program);
        fprintf(stderr, "Couldn't attach vertex shader from file %s\n", vertFileName);
        return 0;
    }

    if (!attachShader(program, GL_FRAGMENT_SHADER, fragFileName))
    {
        glDeleteProgram(program);
        fprintf(stderr, "Couldn't attach fragment shader from file %s\n", fragFileName);
        return 0;
    }

    glLinkProgram(program);
    glGetProgramiv(program, GL_LINK_STATUS, &linked);
```

```
    if (!linked)
    {
        glDeleteProgram(program);
        char temp[256];
        glGetProgramInfoLog(program, 256, 0, temp);
        fprintf(stderr, "Failed to link program: %s\n", temp);
        return 0;
    }

    return program;
}


// ice growth
// generate initial phase and temperature field
void generate_field(float *h_p, float *h_t, int w, int h) {
    int idx = 0;

    for(int j = 0; j < h; j++) {
          for(int i = 0; i < w; i++) {
/*
            if(i == 0) {
                h_p[idx] = 1.0;
                h_t[idx] = 0.0;
            }
            else {
                h_p[idx] = 0.0;
                h_t[idx] = T0;
            }
*/

           if(i * phaseFieldSpacing < growthCellLength) {
                h_t[idx] = T0 + i * phaseFieldSpacing / growthCellLength * (T1 - T0);
                if(h_t[idx] <= 1.0) {
                    h_p[idx] = 1.0;
                }
                else {
                    h_p[idx] = 0.0;
                }
            }
            else {
                h_t[idx] = T1;
                h_p[idx] = 0.0;
            }

            ++idx;
        }
    }
}

void initHostParams(SimParams *params) {
// grid properties
        params->boundary = make_int4(4, 4, 2, 2); // mirror = 0, Neumann   = 1, periodic  = 2,
parallel  = 3, Dirichlet = 4
        params->gridWidth   = gridWidth;
        params->gridHeight  = gridHeight;
        params->T0 = T0;
        params->T1 = T1;

        params->spacing = phaseFieldSpacing; // spacing
        params->worldOrigin = make_float3(-0.5 * gridWidth * phaseFieldSpacing, 0.0f, -0.5 *
gridHeight * phaseFieldSpacing);

// phase field
        params->alpha   = 0.9;
        params->gamma   = 10.0;
        params->epsb    = 0.01;
        params->tao     = 0.0003;
        params->D       = 1.0;
        params->a       = 0.01;
```

```
        params->theta0  = 0.0; //0.5 * M_PI;
        params->Te      = 1.0;
        params->K       = 0;
        params->delta   = 0.05;
        params->j       = 4.0;
// time step
        params->dt      = timeStep;
// particle
        params->gravity = make_float3(-gravity, 0, 0);
        params->particleRadius  = particleRadius;
        params->globalDamping = globalDamping;
        params->particleGridCellSize = 2.0f * particleRadius; // set the cell size to be the
diameter of the particle
        params->particleGridWidth = (int) ceil(0.5f * gridWidth * phaseFieldSpacing /
particleRadius);
        params->particleGridHeight = (int) ceil(0.5f * gridHeight * phaseFieldSpacing /
particleRadius);
        params->numParticleGridCells = params->particleGridWidth * params->particleGridHeight;

        params->collideSpring = collideSpring;
        params->collideDamping = collideDamping;
        params->collideShear = collideShear;
        params->collideAttraction = collideAttraction;
        params->pThreshold = pThreshold;
// slide speed
        params->slideSpeed = slideSpeed;
        params->growthCellLength = growthCellLength;
// others
        params->f = 1.0;
        params->freezeCoeff = freezeCoeff;
        params->tempDepressCoeff = tempDepressCoeff;
}

void initParticles(float *h_p, int w, int h, int n)
{

    float *pos = h_p;

    // generate uniform distribution of particles
    // suppose n = 3 * k^2
    // volumetric density
    int nh = sqrt(n / 3);
//    int nh = 2;
    float s = h * h_params.spacing / nh;
    float initZ = h_params.worldOrigin.z + 0.5 * s;
    // ahead of slider
//    float initX = h_params.worldOrigin.x + growthCellLength + growthCellPos + 0.5 *
h_params.spacing;
    float initX = (0.5 + 0.3 * gridWidth) * h_params.spacing + h_params.worldOrigin.x;
    int k = 0;
    while(k < n) {
        int col = k / nh;
        int row = k - col * nh;
        *pos++ = initX + col * s + (randf() - 0.5f) * 0.5 * s; // jitter
        *pos++ = 0.0f;
        float tmp = initZ + row * s + (randf() - 0.5f) * 0.5 * s;
        if(tmp < h_params.worldOrigin.z) {
            tmp += h * h_params.spacing;
        }
        else if(tmp >= h_params.worldOrigin.z + h * h_params.spacing) {
            tmp -= h * h_params.spacing;
        }
        *pos++ = tmp;
        *pos++ = 1.0f;
        ++k;
    }
}


void initParams()
{
```

```
    paramsGL = new ParamListGL("misc");
    paramsGL->AddParam(new Param<float>("damping coeff", globalDamping, 0.0f, 1.0f, 0.01f,
&globalDamping));
    paramsGL->AddParam(new Param<float>("colloid radius", particleRadius, 0.01f, 0.5f, 0.1f,
&particleRadius));
    paramsGL->AddParam(new Param<float>("hot/cold slide speed", slideSpeed, 0.0f, 32.0f, 8.0f,
&slideSpeed));
    paramsGL->AddParam(new Param<float>("freezeT depress coeff", tempDepressCoeff, 0.0, 2.0f,
0.01f,
                                        &tempDepressCoeff));
    paramsGL->AddParam(new Param<float>("particle freeze coeff", freezeCoeff, 0.0, 1.0f, 0.01f,
&freezeCoeff));
    paramsGL->AddParam(new Param<float>("particle draw size", particleDrawSize, 0.0, 100.0f,
10.0f, &particleDrawSize));
}

void special(int k, int x, int y)
{
    paramsGL->Special(k, x, y);
}


void updateParams(SimParams *params)
{
    params->globalDamping = globalDamping;
    params->particleRadius = particleRadius;
    params->slideSpeed = slideSpeed;
    params->tempDepressCoeff = tempDepressCoeff;
    params->freezeCoeff = freezeCoeff;
    setParameters(params);
}
```

```
// !!! directionalFreezing_kernel.cu
////////////////////////////////////////////////////////////////////////////
// Kernel functions
////////////////////////////////////////////////////////////////////////////

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <GL/glew.h>

#include <cuda_runtime.h>
#include <cuda_gl_interop.h>
#include <curand_kernel.h>

#include <helper_cuda.h>
#include <helper_cuda_gl.h>

#include <helper_functions.h>
#include <math_constants.h>

#include "thrust/device_ptr.h"
#include "thrust/for_each.h"
#include "thrust/iterator/zip_iterator.h"
#include "thrust/sort.h"

#include "phaseField_kernel.cuh"
#include "particles_kernel.cuh"

//Round a / b to nearest higher integer value
int cuda_iDivUp(int a, int b)
{
    return (a + (b - 1)) / b;
}

void computeGridSize(uint n, uint blockSize, uint &numBlocks, uint &numThreads)
{
    numThreads = min(blockSize, n);
    numBlocks = cuda_iDivUp(n, numThreads);
}

// phase-field kernels

__global__ void setup_RNG_kernel(curandState *state, int seed, uint width, uint height)
{
    uint x = blockIdx.x*blockDim.x + threadIdx.x;
    uint y = blockIdx.y*blockDim.y + threadIdx.y;
    uint i = y * width + x;
    if((x < width) && (y < height)) {
          curand_init(seed, i, 0, &state[i]);
    }

}

// wrapper functions
extern "C"
{

void setParameters(SimParams *hostParams)
{
    checkCudaErrors(cudaMemcpyToSymbol(d_params, hostParams, sizeof(SimParams)));
}

void curandInit(curandState *state, uint width, uint height)
{
    dim3 block(16, 16, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    setup_RNG_kernel<<<grid2, block>>>(state, rand(), width, height);

    // check if kernel invocation generated an error
    getLastCudaError("Kernel execution failed: setup_RNG_kernel");
```

```
}


void cudaGenerateHelperFieldKernel(float  *d_phaseMap,
                                   float2 *d_dpdrMap,
                                   float  *d_esqMap,
                                   float  *d_ededthMap,
                                   uint    width,
                                   uint    height)
{
    dim3 block(16, 16, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    generateHelperFieldKernel<<<grid2, block>>>(d_phaseMap, d_dpdrMap, d_esqMap, d_ededthMap,
width, height);
    getLastCudaError("Kernel execution failed: generateHelperFieldKernel");
}


void cudaUpdateFieldmapKernel(float   cellPos,
                              float  *d_phaseMap,
                              float  *d_tempMap,
                              float2 *d_dpdrMap,
                              float  *d_esqMap,
                              float  *d_ededthMap,
                              float  *d_particleDensMap,
                              uint    width,
                              uint    height,
                                   curandState *state)
{
    dim3 block(16, 16, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    updateFieldmapKernel<<<grid2, block>>>(cellPos, d_phaseMap, d_tempMap, d_dpdrMap,
                                           d_esqMap, d_ededthMap, d_particleDensMap, width,
height, state);
    getLastCudaError("Kernel execution failed: updateFieldmapKernel");
}


// particles

void calcHash(uint  *gridParticleHash,
              uint  *gridParticleIndex,
              float *pos,
              int    numParticles)
{
    uint numThreads, numBlocks;
    computeGridSize(numParticles, 256, numBlocks, numThreads);

    // execute the kernel
    calcHashD<<< numBlocks, numThreads >>>(gridParticleHash,
                                           gridParticleIndex,
                                           (float4 *) pos,
                                           numParticles);

    // check if kernel invocation generated an error
    getLastCudaError("Kernel execution failed: clacHashD");
}


void reorderDataAndFindCellStart(uint  *cellStart,
                                 uint  *cellEnd,
                                 float *sortedPos,
                                 float *sortedVel,
                                 uint  *gridParticleHash,
                                 uint  *gridParticleIndex,
                                 float *oldPos,
                                 float *oldVel,
                                 uint   numParticles,
                                 uint   numCells)
{
    uint numThreads, numBlocks;
```

```
    computeGridSize(numParticles, 256, numBlocks, numThreads);

    // set all cells to empty
    checkCudaErrors(cudaMemset(cellStart, 0xffffffff, numCells*sizeof(uint)));

    uint smemSize = sizeof(uint)*(numThreads+1);
    reorderDataAndFindCellStartD<<< numBlocks, numThreads, smemSize>>>(
        cellStart,
        cellEnd,
        (float4 *) sortedPos,
        (float4 *) sortedVel,
        gridParticleHash,
        gridParticleIndex,
        (float4 *) oldPos,
        (float4 *) oldVel,
        numParticles);
    getLastCudaError("Kernel execution failed: reorderDataAndFindCellStartD");

}


void collide(uint  *isParticleFrozen,
             float *newVel,
             float *forceMap,
             float *sortedPos,
             float *sortedVel,
             float2 *distVec,
             uint  *gridParticleIndex,
             uint  *cellStart,
             uint  *cellEnd,
             uint   numParticles,
             uint   numCells)
{
    // thread per particle
    uint numThreads, numBlocks;
    computeGridSize(numParticles, 256, numBlocks, numThreads);

    // execute the kernel
    collideD<<< numBlocks, numThreads >>>(isParticleFrozen,
                                          (float4 *)newVel,
                                          (float3 *)forceMap,
                                          (float4 *)sortedPos,
                                          (float4 *)sortedVel,
                                                    distVec,
                                          gridParticleIndex,
                                          cellStart,
                                          cellEnd,
                                          numParticles);

    // check if kernel invocation generated an error
    getLastCudaError("Kernel execution failed: collideD");
}



void sortParticles(uint *dGridParticleHash, uint *dGridParticleIndex, uint numParticles)
{
    thrust::sort_by_key(thrust::device_ptr<uint>(dGridParticleHash),
                        thrust::device_ptr<uint>(dGridParticleHash + numParticles),
                        thrust::device_ptr<uint>(dGridParticleIndex));
}


void cudaUpdateParticleKernel(uint  *isParticleFrozen,
                              float *particlePos,
                                   float *particleVel,
                                   uint   numParticles,
                              float *phaseMap,
                              uint   width,
                              uint   height)
{
```

```
    uint numThreads, numBlocks;
    computeGridSize(numParticles, 256, numBlocks, numThreads);
    updateParticleKernel<<<numBlocks, numThreads>>>(isParticleFrozen, (float4 *) particlePos,
(float4 *) particleVel, numParticles,
                                                    phaseMap, width, height);
    getLastCudaError("Kernel execution failed: updateParticleKernel");
}

void cudaUpdateParticleFrozenKernel(uint   *isParticleFrozen,
                                    float  *particlePos,
                                    uint    numParticles,
                                    float  *phaseMap,
                                    uint    width,
                                    uint    height)
{
    uint numThreads, numBlocks;
    computeGridSize(numParticles, 256, numBlocks, numThreads);
    updateParticleFrozenKernel<<<numBlocks, numThreads>>>(isParticleFrozen, (float4 *)
particlePos, numParticles,
                                   phaseMap, width, height);
    getLastCudaError("Kernel execution failed: updateParticleKernel");
}



void cudaCalculateParticleVolumeKernel(uint  *isParticleFrozen,
                                       float *particlePos,
                                       float *phaseMap,
                                       float *particleVolumeMap,
                                       float  radius,
                                       uint   width,
                                       uint   height,
                                       uint   numParticles)
{
    uint numThreads, numBlocks;
    computeGridSize(numParticles, 256, numBlocks, numThreads);
    calculateParticleVolumeKernel<<<numBlocks, numThreads>>>(isParticleFrozen, (float4 *)
particlePos, phaseMap, particleVolumeMap,
                                       radius, width, height, numParticles);
    getLastCudaError("Kernel execution failed: calculateParticleVolumeKernel");
}


void cudaCalculateLiquidVolumeKernel(float  *phaseMap,
                                     float  *liquidVolumeMap,
                                     float  radius,
                                     uint   width,
                                     uint   height)
{
    dim3 block(16, 16, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    calculateLiquidVolumeKernel<<<grid2, block>>>(phaseMap, liquidVolumeMap, radius, width,
height);
    getLastCudaError("Kernel execution failed: calculateLiquidVolumeKernel");

}

void cudaCalculateParticleDensityKernel(float  *particleVolumeMap,
                                        float  *liquidVolumeMap,
                                        float  *particleDensMap,
                                        uint    width,
                                        uint    height)
{
    dim3 block(16, 16, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    calculateParticleDensityKernel<<<grid2, block>>>(particleVolumeMap, liquidVolumeMap,
particleDensMap, width, height);
    getLastCudaError("Kernel execution failed: calculateParticleDensityKernel");

}
```

```
void cudaCheckInterfaceKernel(int  *isInterface,
                              float *phaseMap,
                              uint   width,
                              uint   height)
{
    dim3 block(16, 16, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    checkInterfaceKernel<<<grid2, block>>>(isInterface, phaseMap, width, height);
    getLastCudaError("Kernel execution failed: checkInterfaceKernel");

}

void cudaCalculateSignedDistanceKernel(float  *signedDist,
                                       float2 *distVec,
                                       int   *isInterface,
                                       uint   width,
                                       uint   height)
{
    dim3 block(16, 16, 1);
    dim3 grid2(cuda_iDivUp(width, block.x), cuda_iDivUp(height, block.y), 1);
    calculateSignedDistanceKernel<<<grid2, block>>>(signedDist, distVec, isInterface, width,
height);
    getLastCudaError("Kernel execution failed: calculateSignedDistanceKernel");

}


} // extern "C"
```

```
// !!! particles_kerne.cuh
////////////////////////////////////////////////////////////////////////////////
// device functions for particle simulation
////////////////////////////////////////////////////////////////////////////////
#ifndef PARTICLES_KERNEL_H
#define PARTICLES_KERNEL_H

#include "vector_types.h"
#include <cstdio>
#include <cmath>
#include "helper_math.h"
#include "math_constants.h"
#include "parameters.h"
#include "phaseField_kernel.cuh"

__device__ int3 calcParticleGridPos(float3 p)
{
    int3 gridPos;
    gridPos.x = floor((p.x - d_params.worldOrigin.x) / d_params.particleGridCellSize);
    gridPos.y = 0;
    gridPos.z = floor((p.z - d_params.worldOrigin.z) / d_params.particleGridCellSize);
    return gridPos;
}

__device__ int3 calcPhaseGridPos(float3 p)
{
    int3 gridPos;
    gridPos.x = floor((p.x - d_params.worldOrigin.x) / d_params.spacing);
    gridPos.y = 0;
    gridPos.z = floor((p.z - d_params.worldOrigin.z) / d_params.spacing);
    return gridPos;
}

__device__ uint calcGridHash(int3 gridPos)
{
    return gridPos.z * d_params.particleGridWidth + gridPos.x;
}

__global__ void calcHashD(uint    *gridParticleHash,
                          uint    *gridParticleIndex,
                          float4  *pos,
                          uint     numParticles)
{
    uint index = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    if(index >= numParticles) return;
    volatile float4 p = pos[index];
    int3 gridPos = calcParticleGridPos(make_float3(p));
    uint hash = calcGridHash(gridPos);

    gridParticleHash[index] = hash;
    gridParticleIndex[index] = index;
}


__global__
void reorderDataAndFindCellStartD(uint   *cellStart,        // output: cell start index
                                  uint   *cellEnd,          // output: cell end index
                                  float4 *sortedPos,        // output: sorted positions
                                  float4 *sortedVel,        // output: sorted velocities
                                  uint   *gridParticleHash, // input: sorted grid hashes
                                  uint   *gridParticleIndex,// input: sorted particle indices
                                  float4 *oldPos,           // input: sorted position array
                                  float4 *oldVel,           // input: sorted velocity array
                                  uint    numParticles)
{
    extern __shared__ uint sharedHash[];    // blockSize + 1 elements
    uint index = __umul24(blockIdx.x,blockDim.x) + threadIdx.x;

    uint hash;

    // handle case when no. of particles not multiple of block size
```

```
    if (index < numParticles)
    {
        hash = gridParticleHash[index];

        // Load hash data into shared memory so that we can look
        // at neighboring particle's hash value without loading
        // two hash values per thread
        sharedHash[threadIdx.x+1] = hash;

        if (index > 0 && threadIdx.x == 0)
        {
            // first thread in block must load neighbor particle hash
            sharedHash[0] = gridParticleHash[index-1];
        }
    }

    __syncthreads();

    if (index < numParticles)
    {
        // If this particle has a different cell index to the previous
        // particle then it must be the first particle in the cell,
        // so store the index of this particle in the cell.
        // As it isn't the first particle, it must also be the cell end of
        // the previous particle's cell

        if (index == 0 || hash != sharedHash[threadIdx.x])
        {
            cellStart[hash] = index;

            if (index > 0)
                cellEnd[sharedHash[threadIdx.x]] = index;
        }

        if (index == numParticles - 1)
        {
            cellEnd[hash] = index + 1;
        }

        // Now use the sorted index to reorder the pos and vel data
        uint sortedIndex = gridParticleIndex[index];
        float4 pos = oldPos[sortedIndex];
        float4 vel = oldVel[sortedIndex];

        sortedPos[index] = pos;
        sortedVel[index] = vel;
    }


}


// collide two spheres using DEM method
__device__
float3 collideSpheres(float3 posA, float3 posB,
                      float3 velA, float3 velB,
                      float radiusA, float radiusB)
{
    float3 force = make_float3(0.0f);
    // calculate relative position
    float3 relPos = posB - posA;
// apply PBC in z
    float h = d_params.spacing * d_params.gridHeight;
    if(relPos.z > 0.5 * h) {
        relPos.z = relPos.z - h;
    }
    else if(relPos.z < -0.5 * h) {
        relPos.z = relPos.z + h;
    }

    float dist = length(relPos);
```

```
//    float collideDist = radiusA + radiusB;
    // add a thin layer so that the solidification can enter the pores
    float collideDist = 1.05 * (radiusA + radiusB);

    if (dist < collideDist)
    {
        float3 norm = relPos / dist;

        // relative velocity
        float3 relVel = velB - velA;

        // relative tangential velocity
        float3 tanVel = relVel - (dot(relVel, norm) * norm);

        // spring force
//        force = -d_params.collideSpring*(collideDist - dist) * norm;
        // much higher than between interface and sphere
        force = -10.0 * d_params.collideSpring*(collideDist - dist) * norm;

        // dashpot (damping) force
        force += d_params.collideDamping*relVel;
        // tangential shear force
        force += d_params.collideShear*tanVel;
/*
        // attraction
        force += d_params.collideAttraction*relPos;
*/
    }

    return force;
}


// collide a particle against all other particles in a given cell
__device__
float3 collideParticleCell(int3    gridPos,
                       uint    index,
                            float3  pos,
                            float3  vel,
                            float4 *oldPos,
                            float4 *oldVel,
                            uint   *cellStart,
                            uint   *cellEnd)
{
    uint gridHash = calcGridHash(gridPos);

    // get start of bucket for this cell
    uint startIndex = cellStart[gridHash];

    float3 force = make_float3(0.0f);

    if (startIndex != 0xffffffff)          // cell is not empty
    {
        // iterate over particles in this cell
        uint endIndex = cellEnd[gridHash];

        for (uint j=startIndex; j<endIndex; j++)
        {
            if (j != index)                // check not colliding with self
            {
                float3 pos2 = make_float3(oldPos[j]);
                float3 vel2 = make_float3(oldVel[j]);

                // collide two spheres
                force += collideSpheres(pos, pos2, vel, vel2, d_params.particleRadius,
d_params.particleRadius);
            }
        }
    }

    return force;
```

```
}


__device__
float3 collideParticleGrid(uint    index,
                           float3  pos,
                           float3  vel,
                           float4 *oldPos,
                           float4 *oldVel,
                           uint   *cellStart,
                           uint   *cellEnd)
{
    // get address in grid
    int3 gridPos = calcParticleGridPos(pos);

    // examine neighboring cells
    float3 force = make_float3(0.0f);

    for (int z=-1; z<=1; z++)
    {
        for (int x=-1; x<=1; x++)
        {
            int3 neighborPos = gridPos + make_int3(x, 0, z);
            if(neighborPos.x >= 0 && neighborPos.x < d_params.particleGridWidth) {
                check_boundary(neighborPos.z, 0, d_params.particleGridHeight, d_params.boundary.z,
d_params.boundary.w);
                force += collideParticleCell(neighborPos, index, pos, vel, oldPos, oldVel,
cellStart, cellEnd);
            }
        }
    }
    return force;
}


__device__ float3 collidePhasePoint(float3   particlePos,
                                    float3   particleVel,
                                    float    particleRadius,
                                    float3   phasePos)
{
    // calculate relative position
    float3 relPos = phasePos - particlePos;
// apply PBC in z
    float h = d_params.spacing * d_params.gridHeight;
    if(relPos.z > 0.5 * h) {
        relPos.z = relPos.z - h;
    }
    else if(relPos.z < -0.5 * h) {
        relPos.z = relPos.z + h;
    }

    float dist = length(relPos);
    float collideDist = particleRadius + d_params.spacing;

    float3 force = make_float3(0.0f);

    if (dist < collideDist)
    {
        float3 norm = relPos / dist;

        // relative velocity
        float3 relVel = -particleVel;

        // relative tangential velocity
        float3 tanVel = relVel - (dot(relVel, norm) * norm);

        // spring force
        force = -d_params.collideSpring*(collideDist - dist) * norm;
        // dashpot (damping) force
        force += d_params.collideDamping*relVel;
        // tangential shear force
```

```
        force += d_params.collideShear*tanVel;
        // attraction
        force += d_params.collideAttraction*relPos;
    }

    return force;
}


__device__ float2 bilinear(float3 particlePos,
                           float2 *distVec)
{
    int3 gp = calcPhaseGridPos(particlePos);
    float2 v00 = fetch(distVec, make_int2(gp.x, gp.z));
    float2 v10 = fetch(distVec, make_int2(gp.x + 1, gp.z));
    float2 v01 = fetch(distVec, make_int2(gp.x, gp.z + 1));
    float2 v11 = fetch(distVec, make_int2(gp.x + 1, gp.z + 1));
    float x0 = gp.x * d_params.spacing + d_params.worldOrigin.x;
    float y0 = gp.z * d_params.spacing + d_params.worldOrigin.z;
//    float x1 = x0 + d_params.spacing;
//    float y1 = y0 + d_params.spacing;

//    float xd = (particlePos.x - x0) / (x1 - x0);
//    float yd = (particlePos.z - y0) / (y1 - y0);

    float xd = (particlePos.x - x0) / d_params.spacing;
    float yd = (particlePos.z - y0) / d_params.spacing;

    float2 v0 = v00 * (1 - xd) + v10 * xd;
    float2 v1 = v01 * (1 - xd) + v11 * xd;

    return v0 * (1 - yd) + v1 * yd;
}



__device__ float3 collideInterface(float3 particlePos,
                                   float3 particleVel,
                                   float  particleRadius,
                                   float2  *distVec)
{
   // calculate relative position
    float2 r = bilinear(particlePos, distVec);
    float3 relPos = make_float3(-r.x, 0.0f, -r.y);

    float dist = length(relPos);
    float collideDist = 1.2 * particleRadius;

    float3 force = make_float3(0.0f);

    if (dist < collideDist)
    {
        float3 norm = relPos / dist;

        // relative velocity
        float3 relVel = -particleVel;

        // relative tangential velocity
        float3 tanVel = relVel - (dot(relVel, norm) * norm);

        // spring force
        force = -d_params.collideSpring*(collideDist - dist) * norm;
        // dashpot (damping) force
        force += d_params.collideDamping*relVel;
        // tangential shear force
        force += d_params.collideShear*tanVel;
        // attraction
        force += d_params.collideAttraction*relPos;
    }

    return force;
```

```
}


// collid a particle with all other phase points in a given cell
__device__ float3 collidePhaseCell(float3  particlePos,
                                   float3  particleVel,
                                       float  *phaseMap,
                                   uint    width,
                                   uint    height,
                                   float   pThreshold)
{
          float3 force = make_float3(0.0f);
        float3 phasePos = make_float3(0.0f);
        int3 nearestPhaseGridPos = calcPhaseGridPos(particlePos);
        int cellSize = 3;
        int neighborX, neighborZ;
        for(int i = -cellSize; i <= cellSize; i++) {
             neighborX = nearestPhaseGridPos.x + i;
            if(neighborX < 0 || neighborX >= width) {
                continue;
            }
              for(int j = -cellSize; j <= cellSize; j++) {
                neighborZ = nearestPhaseGridPos.z + j;
                check_boundary(neighborZ, 0, height, d_params.boundary.z, d_params.boundary.w);
                   int idx = neighborZ * width + neighborX;
                if(phaseMap[idx] >= pThreshold) {
                    phasePos = make_float3(neighborX * d_params.spacing + d_params.worldOrigin.x,
0.0f,
                                           neighborZ * d_params.spacing + d_params.worldOrigin.z);
                    force += phaseMap[idx] * collidePhasePoint(particlePos, particleVel,
d_params.particleRadius, phasePos);
                }
              }
        }
        return force;
}


__global__
void collideD(uint   *isParticleFrozen,
              float4 *newVel,               // output: new velocity
              float3 *forceMap,             // output: force map
              float4 *oldPos,               // input: sorted positions
              float4 *oldVel,               // input: sorted velocities
              float2  *distVec,        // input: distance vector field
              uint   *gridParticleIndex,    // input: sorted particle indices
              uint   *cellStart,
              uint   *cellEnd,
              uint    numParticles)
{
    uint index = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;

    if (index >= numParticles) return;
    uint originalIndex = gridParticleIndex[index];
    if(isParticleFrozen[originalIndex] == 1) {
        forceMap[originalIndex] = make_float3(0.0f);
        return;
    }

    // read particle data from sorted arrays
    float3 pos = make_float3(oldPos[index]);
    float3 vel = make_float3(oldVel[index]);
    float3 force = make_float3(0.0f);

//    // force with phase-field
//    force += collidePhaseCell(pos, vel, phaseMap, d_params.gridWidth, d_params.gridHeight,
d_params.pThreshold);
    // force with interface
    force += collideInterface(pos, vel, d_params.particleRadius, distVec);
    // force with other particles
```

```
    force += collideParticleGrid(index, pos, vel, oldPos, oldVel, cellStart, cellEnd);

    // write new velocity back to original unsorted location
    newVel[originalIndex] = make_float4(vel + force, 0.0f);
    forceMap[originalIndex] = force;
}


__global__ void updateParticleKernel(uint   *isParticleFrozen,
                                     float4 *particlePos,
                                     float4 *particleVel,
                                     uint    nParticles,
                                     float  *phaseMap,
                                     uint    width,
                                     uint    height)
{
    uint index = __mul24(blockIdx.x, blockDim.x) + threadIdx.x;
    if(index >= nParticles) return;
    if(isParticleFrozen[index] == 1) return;

    float3 vel = make_float3(particleVel[index]);
    float3 pos = make_float3(particlePos[index]);
    vel += d_params.gravity * d_params.dt;
    vel *= d_params.globalDamping;
    pos += vel * d_params.dt;

// boundary
    float xlen = width * d_params.spacing;
    float zlen = height * d_params.spacing;
    if(pos.x >=  d_params.worldOrigin.x + d_params.particleRadius &&
       pos.x < d_params.worldOrigin.x + xlen - d_params.particleRadius) {
        particlePos[index].x = pos.x;
    }
    if(pos.z < d_params.worldOrigin.z) {
        particlePos[index].z = pos.z + zlen;
    }
    else if(pos.z >= d_params.worldOrigin.z + zlen) {
        particlePos[index].z = pos.z - zlen;
    }
    else {
        particlePos[index].z = pos.z;
    }
}

__global__ void updateParticleFrozenKernel(uint   *isParticleFrozen,
                                           float4 *particlePos,
                                           uint    nParticles,
                                           float  *phaseMap,
                                           uint    width,
                                           uint    height)
{
    // check if the particle is surrounded by more solid than liquid
    float r0 = d_params.particleRadius;
    float r1 = 1.25 * r0;
    uint index = __mul24(blockIdx.x, blockDim.x) + threadIdx.x;
    if(index >= nParticles) return;
    if(isParticleFrozen[index] == 1) return;

    int cellSize = r1 / d_params.spacing + 1;

    float3 pos = make_float3(particlePos[index]);
    int3 nearestPhaseGridPos = calcPhaseGridPos(pos);
    float dx, dz;
    int neighborX, neighborZ;

    float amtSld = 0.0;
    float amtTot = 0.0;
    float h = d_params.spacing * d_params.gridHeight;
    for(int i = -cellSize; i <= cellSize; i++) {
        neighborX = nearestPhaseGridPos.x + i;
        if(neighborX >= 0 && neighborX < width) {
```

```
            for(int j = -cellSize; j <= cellSize; j++) {
                neighborZ = nearestPhaseGridPos.z + j;
                check_boundary(neighborZ, 0, height, d_params.boundary.z, d_params.boundary.w);
                dx = pos.x - neighborX * d_params.spacing - d_params.worldOrigin.x;
                dz = pos.z - neighborZ * d_params.spacing - d_params.worldOrigin.z;
// apply PBC in z
                if(dz > 0.5 * h) {
                    dz -= h;
                }
                else if(dz < -0.5 * h) {
                    dz += h;
                }
                float drq = dx * dx + dz * dz;
                if(drq > r0 * r0 && drq < r1 * r1) {
                    amtTot += 1.0;
                    uint gridIndex = neighborZ * width + neighborX;
                    if(phaseMap[gridIndex] >= d_params.pThreshold) {
                        amtSld += 1.0;
                    }
                }
            }
        }
    }
    if(amtTot > 0.5 && amtSld / amtTot > 0.5) {
        isParticleFrozen[index] = 1;
    }
    else {
        isParticleFrozen[index] = 0;
    }
}


__global__ void calculateParticleVolumeKernel(uint     *isParticleFrozen,
                                               float4   *particlePos,
                                               float    *phaseMap,
                                               float    *particleVolumeMap,
                                               float     radius,
                                               float     width,
                                               float     height,
                                               float     nParticles)
{
    uint index = __mul24(blockIdx.x, blockDim.x) + threadIdx.x;
    if(index >= nParticles) return;
//    if(isParticleFrozen[index] == 1) return;

    int cellSize = ceil(radius / d_params.spacing) + 1;
    float3 pos = make_float3(particlePos[index]);
    int3 nearestPhaseGridPos = calcPhaseGridPos(pos);
    float dx, dz;
    int neighborX, neighborZ;

    for(int i = -cellSize; i <= cellSize; i++) {
        for(int j = -cellSize; j <= cellSize; j++) {
            neighborX = nearestPhaseGridPos.x + i;
            if(neighborX >= 0 && neighborX < width) {
                neighborZ = nearestPhaseGridPos.z + j;
                dx = pos.x - neighborX * d_params.spacing - d_params.worldOrigin.x;
                dz = pos.z - neighborZ * d_params.spacing - d_params.worldOrigin.z;
                check_boundary(neighborZ, 0, height, d_params.boundary.z, d_params.boundary.w);
                if(dx * dx + dz * dz <= radius * radius) {
                    uint gridIndex = neighborZ * width + neighborX;
//                      if(phaseMap[gridIndex] < d_params.pThreshold) {
                        particleVolumeMap[gridIndex] = 1.0;
//                      }
                }
            }
        }
    }
}

#endif // PARTICLES_KERNEL_H
```

```
// !!! phaseField_kernel.cuh
////////////////////////////////////////////////////////////////////////////////
// device functions for phase field
////////////////////////////////////////////////////////////////////////////////

#ifndef PHASEFIELD_KERNEL_H
#define PHASEFIELD_KERNEL_H

#include "vector_types.h"
#include <cstdio>
#include <cmath>
#include "helper_math.h"
#include "math_constants.h"
#include "parameters.h"

__constant__ SimParams d_params;

__device__ int2 position(int k)
{
        int j = k / d_params.gridWidth;
        int i = k % d_params.gridWidth;
        return make_int2(i, j);
}

__device__ void check_boundary(int& x, int x0, int x1, int b0, int b1)
{
        if(x < x0) {
                if(b0 == 1 || b0 == 4) {
                    x = x0;
                }
                else if(b0 == 2) {
                    x = x + x1 - x0;
                }
                else if(b0 == 0) {
                    x = 2 * x0 - x;
                }
        }
        else if(x >= x1) {
                if(b1 == 1 || b1 == 4) {
                    x = x1 - 1;
                }
                else if(b1 == 2) {
                    x = x - (x1 - x0);
                }
                else if(b1 == 0) {
                    x = 2 * (x1 - 1) - x;
                }
        }
}

__device__ bool isDirichlet(int x, int x0, int x1, int b0, int b1)
{
    if(((x == x0) && (b0 == 4)) || ((x == x1 - 1) && (b1 == 4)))
          return true;
    else
          return false;
}

__device__ float fetch(float *field, int2 pos)
{
        int2 p = pos;
        check_boundary(p.x, 0, d_params.gridWidth, d_params.boundary.x, d_params.boundary.y);
        check_boundary(p.y, 0, d_params.gridHeight, d_params.boundary.z, d_params.boundary.w);
        int k = p.y * d_params.gridWidth + p.x;
        return field[k];
}

// fetch element from multi-field, idx = 0 or 1
__device__ float fetch(float2 *field, int2 pos, int idx)
{
        int2 p = pos;
```

```
        check_boundary(p.x, 0, d_params.gridWidth, d_params.boundary.x, d_params.boundary.y);
        check_boundary(p.y, 0, d_params.gridHeight, d_params.boundary.z, d_params.boundary.w);
        int k = p.y * d_params.gridWidth + p.x;
        if(idx == 0) {
                return field[k].x;
        }
        else {
                return field[k].y;
        }
}

__device__ float2 fetch(float2 *field, int2 pos)
{
        int2 p = pos;
        check_boundary(p.x, 0, d_params.gridWidth, d_params.boundary.x, d_params.boundary.y);
        check_boundary(p.y, 0, d_params.gridHeight, d_params.boundary.z, d_params.boundary.w);
        int k = p.y * d_params.gridWidth + p.x;
        return field[k];
}

__device__ float2 gradient(float *field, int2 pos)
{
        float2 grad;
        float weight = 1.0 / (2.0 * d_params.spacing);
        grad.x = (fetch(field, make_int2(pos.x + 1, pos.y)) - fetch(field, make_int2(pos.x - 1,
pos.y))) * weight;
        grad.y = (fetch(field, make_int2(pos.x, pos.y + 1)) - fetch(field, make_int2(pos.x, pos.y
- 1))) * weight;
        return grad;
}

__device__ float divergence(float2 *field, int2 pos)
{
        float div = 0;
        float weight = 1.0 / (2.0 * d_params.spacing);
        div += (fetch(field, make_int2(pos.x + 1, pos.y), 0) - fetch(field, make_int2(pos.x - 1,
pos.y), 0)) * weight;
        div += (fetch(field, make_int2(pos.x, pos.y + 1), 1) - fetch(field, make_int2(pos.x,
pos.y - 1), 1)) * weight;
        return div;
}


__device__ float laplacian(float *field, int2 pos)
{
        float lap = 0;
        float weight = 1.0 / (d_params.spacing * d_params.spacing);
        lap = 2.0f *
             (fetch(field, make_int2(pos.x + 1, pos.y)) + fetch(field, make_int2(pos.x - 1,
pos.y))
            + fetch(field, make_int2(pos.x, pos.y + 1)) + fetch(field, make_int2(pos.x, pos.y -
1))
            - 4.0 * fetch(field, pos))
            + 0.5f *
              (fetch(field, make_int2(pos.x + 1, pos.y + 1)) + fetch(field, make_int2(pos.x + 1,
pos.y - 1))
            + fetch(field, make_int2(pos.x - 1, pos.y + 1)) + fetch(field, make_int2(pos.x - 1,
pos.y - 1))
            - 4.0 * fetch(field, pos));
        lap = lap / 3.0f * weight;
        return lap;
}

__device__ float curvature(float *field, int2 pos)
{
        float small = 0.001;
          float curv = 0.0f;
        float weight = 1.0 / d_params.spacing;
        float df1, df2;
        for(int i = 0; i <= 1; i++) {
```

```
          df1 = fetch(field, make_int2(pos.x + i, pos.y)) - fetch(field, make_int2(pos.x + i - 1,
pos.y));
                df2 = fetch(field, make_int2(pos.x + i - 1, pos.y + 1)) + fetch(field,
make_int2(pos.x + i, pos.y + 1))
                    - fetch(field, make_int2(pos.x + i - 1, pos.y - 1)) - fetch(field,
make_int2(pos.x + i, pos.y - 1));
                if(abs(df1) > small || abs(df2) > small) {
                   curv += (2 * i - 1) * df1 / sqrt(df1 * df1 + 0.0625 * df2 * df2) * weight;
                }
                df1 = fetch(field, make_int2(pos.x, pos.y + i)) - fetch(field, make_int2(pos.x,
pos.y + i - 1));
                df2 = fetch(field, make_int2(pos.x + 1, pos.y + i - 1)) + fetch(field,
make_int2(pos.x + 1, pos.y + i))
                    - fetch(field, make_int2(pos.x - 1, pos.y + i - 1)) - fetch(field,
make_int2(pos.x - 1, pos.y + i));
                if(abs(df1) > small || abs(df2) > small) {
                        curv += (2 * i - 1) * df1 / sqrt(df1 * df1 + 0.0625 * df2 * df2) * weight;
                }
        }
        return curv;
}


// generate intermediate arrays
__global__ void generateHelperFieldKernel(float  *d_phaseMap,
                                          float2 *d_dpdrMap,
                                          float  *d_esqMap,
                                          float  *d_ededthMap,
                                          uint width,
                                          uint height)
{
    float small = 0.001;
    uint x = blockIdx.x*blockDim.x + threadIdx.x;
    uint y = blockIdx.y*blockDim.y + threadIdx.y;
    if(x >= width || y >= height) return;
    uint index = y * width + x;
    int2 pos = make_int2(x, y);
    float2 dpdr = gradient(d_phaseMap, pos);
    float theta = 0.0f;
    if(abs(dpdr.x) > small || abs(dpdr.y) > small) {
        float dxdr = -dpdr.x / sqrt(dpdr.x * dpdr.x + dpdr.y * dpdr.y);
        dxdr = clamp(dxdr, -1.0f, 1.0f);
        theta = acos(dxdr);
        if(dpdr.y > 0) {
            theta = -theta;
        }
    }
    float jth = d_params.j * (theta - d_params.theta0);
    float eps = d_params.epsb * (1.0 + d_params.delta * cos(jth));
    d_esqMap[index] = eps * eps;
    d_ededthMap[index] = -eps * d_params.epsb * d_params.j * d_params.delta * sin(jth);
    d_dpdrMap[index] = dpdr;
}

// update field
__global__ void updateFieldmapKernel(float   cellPos,
                                     float  *d_phaseMap,
                                     float  *d_tempMap,
                                     float2 *d_dpdrMap,
                                     float  *d_esqMap,
                                     float  *d_ededthMap,
                                     float  *d_particleDensMap,
                                     uint width,
                                     uint height,
                                     curandState *state)
{

    uint x = blockIdx.x*blockDim.x + threadIdx.x;
    uint y = blockIdx.y*blockDim.y + threadIdx.y;
    if(x >= width || y >= height) return;
    uint index = y * width + x;
```

```
    int2 pos = make_int2(x, y);
// check if Dirichlet boundary
    if(isDirichlet(x, 0, width, d_params.boundary.x, d_params.boundary.y) ||
       isDirichlet(y, 0, height, d_params.boundary.z, d_params.boundary.w))
    {
        return;
    }
    else {

        float p = d_phaseMap[index];
        float temp = d_tempMap[index];
        float2 dpdr = d_dpdrMap[index];

        if(d_particleDensMap[index] > 0.9) {
            d_particleDensMap[index] = 0.9;
        }
        float Tm = d_params.Te - d_params.tempDepressCoeff * tan(1.7 * d_particleDensMap[index]);
// if use number of particles
//        float Tm = d_params.Te - d_params.tempDepressCoeff * tan(0.4 *
d_particleDensMap[index]);

//        float Tm = d_params.Te;
        float m = d_params.alpha / M_PI * atan(d_params.gamma * (Tm - temp));
        float2 lap;
        lap.x = laplacian(d_phaseMap, pos);
        lap.y = laplacian(d_tempMap, pos);
        float2 desqdr = gradient(d_esqMap, pos);
        float2 dededthdr = gradient(d_ededthMap, pos);
        float dpdt = 1.0 / d_params.tao * (-dededthdr.x * dpdr.y + dededthdr.y * dpdr.x
                   + desqdr.x * dpdr.x + desqdr.y * dpdr.y + d_esqMap[index] * lap.x
                   + p * (1.0 - p) * (p - 0.5 + m)
                   + d_params.a * p * (1.0 - p) * (curand_uniform(&state[index]) - 0.5));
        d_phaseMap[index] = p + dpdt * d_params.dt;

        float posX = x * d_params.spacing;
        if(posX < cellPos + d_params.growthCellLength) {
            d_tempMap[index] = temp + d_params.dt * (d_params.D * lap.y + d_params.K * dpdt
                             - (d_params.T1 - d_params.T0) / d_params.growthCellLength *
d_params.slideSpeed);
        }
        else {
          d_tempMap[index] = temp + d_params.dt * (d_params.D * lap.y + d_params.K * dpdt);
        }

//        d_tempMap[index] = temp + d_params.dt * (d_params.D * lap.y + d_params.K * dpdt);
    }
}


__global__ void calculateLiquidVolumeKernel(float  *phaseMap,
                                            float  *liquidVolumeMap,
                                            float   radius,
                                            uint    width,
                                            uint    height)
{
    uint x = blockIdx.x*blockDim.x + threadIdx.x;
    uint y = blockIdx.y*blockDim.y + threadIdx.y;
    if(x >= width || y >= height) return;
    uint index = y * width + x;
/*
// ignore solid phase
    if(phaseMap[index] >= d_params.pThreshold) {
        liquidVolumeMap[index] = 0.0f;
        return;
    }
// scan the square area
    int neighborX, neighborY;
    int r = ceil(radius / d_params.spacing);
    for(int i = -r - 1; i <= r + 1; i++) {
        neighborX = x + i;
        if(neighborX >= 0 && neighborX < width) {
```

```
            for(int j = -r - 1; j <= r + 1; j++) {
                neighborY = y + j;
                float dx = i * d_params.spacing;
                float dy = j * d_params.spacing;
                check_boundary(neighborY, 0, height, d_params.boundary.z, d_params.boundary.w);
                if(dx * dx + dy * dy <= radius * radius && phaseMap[neighborY * width + neighborX]
< d_params.pThreshold) {
                    liquidVolumeMap[index] += 1.0f;
                }
            }
        }
    }
*/
    liquidVolumeMap[index] = 1.0f;

}


__global__ void calculateParticleDensityKernel(float  *particleVolumeMap,
                                                float  *liquidVolumeMap,
                                                float  *particleDensMap,
                                                uint    width,
                                                uint    height)
{
    uint x = blockIdx.x*blockDim.x + threadIdx.x;
    uint y = blockIdx.y*blockDim.y + threadIdx.y;
    if(x >= width || y >= height) return;
    uint index = y * width + x;
/*
    if(liquidVolumeMap[index] > 0) {
        particleDensMap[index] = particleVolumeMap[index] / liquidVolumeMap[index]
                               * M_PI * d_params.particleRadius * d_params.particleRadius /
d_params.spacing / d_params.spacing;
//        particleDensMap[index] = particleVolumeMap[index];
    }
    else {
        particleDensMap[index] = 0.0f;
    }
*/
    particleDensMap[index] = particleVolumeMap[index] / liquidVolumeMap[index] * 0.9;
}

__global__ void checkInterfaceKernel(int  *isInterface,
                                     float *phaseMap,
                                     uint  width,
                                     uint  height)
{
    uint x = blockIdx.x*blockDim.x + threadIdx.x;
    uint y = blockIdx.y*blockDim.y + threadIdx.y;
    if(x >= width || y >= height) return;
    uint index = y * width + x;
    // check four corners of the grid
    int f1 = (fetch(phaseMap, make_int2(x, y)) >= d_params.pThreshold);
    int f2 = (fetch(phaseMap, make_int2(x+1, y)) >= d_params.pThreshold);
    int f3 = (fetch(phaseMap, make_int2(x, y+1)) >= d_params.pThreshold);
    int f4 = (fetch(phaseMap, make_int2(x+1, y+1)) >= d_params.pThreshold);
    if((f1 == f2) && (f2 == f3) && (f3 == f4)) {
        if(f1 == 0) {
            isInterface[index] = 1; // liquid
        }
        else {
            isInterface[index] = -1;// solid
        }
    }
    else {
        isInterface[index] = 0; // interface
    }
}

__global__ void calculateSignedDistanceKernel(float  *signedDist,
                                               float2 *distVec,
```

```
                                            int    *isInterface,
                                            uint width,
                                            uint height)
{
    int nnb = 20;
    uint x = blockIdx.x*blockDim.x + threadIdx.x;
    uint y = blockIdx.y*blockDim.y + threadIdx.y;
    if(x >= width || y >= height) return;
    uint index = y * width + x;

    if(isInterface[index] == 0) {
        signedDist[index] = 0;
        distVec[index] = make_float2(0.0);
        return;
    }
    float sdq = 400.0;
    int cx, cy;
    // distance vector
    int mx = width;
    int my = height;
    for(int i = -nnb; i <= nnb; ++i) {
        cx = x + i;
        if(cx < 0 || cx >= width) continue;
        for(int j = -nnb; j <= nnb; ++j) {
            cy = y + j;
            check_boundary(cy, 0, height, d_params.boundary.z, d_params.boundary.w);
            int idx = cy * width + cx;
            if(isInterface[idx] == 0) {
                float sijq = i * i + j * j;
                if(sijq < sdq) {
                    sdq = sijq;
                    mx = i;
                    my = j;
                }
            }
        }
    }
    signedDist[index] = isInterface[index] * sqrt(sdq) * d_params.spacing;
    distVec[index] = make_float2(-mx, -my) * d_params.spacing;
}


#endif //PHASEFIELD_KERNEL_H
```

```
// !!! ./data/particle.frag
////////////////////////////////////////////////////////////////////////////
// GLSL fragment shader for particles
////////////////////////////////////////////////////////////////////////////

varying vec3 force;
varying vec4 pos;

void main()
{
    float PI = 3.1415927;
    float fl = length(force);
    float scale = 2.0 / PI * atan(fl);
    vec4 colorBeg = vec4(0.5, 0.5, 0.5, 0);
    vec4 colorEnd = vec4(1.0, 1.0, 1.0, 0);
/*
    if(scale < 0.0) {
        gl_FragColor = colorBeg;
    }
    else if(scale > 1.0) {
        gl_FragColor = colorEnd;
    }
    else {
        gl_FragColor = (1.0 - scale) * colorBeg + scale * colorEnd;
    }
*/
    vec2 dist = gl_FragCoord.xz - vec2(pos.x, pos.z);
    float dsq = dot(dist, dist);
    if(dsq <= 100.0)
        gl_FragColor = vec4(0.0, 0.0, 0.0, 0.0);
}
```

```
// !!! ./data/phaseField.frag
////////////////////////////////////////////////////////////////////////////////
// GLSL fragment shader for phase field
////////////////////////////////////////////////////////////////////////////////

varying float field;

uniform int chosenField;
uniform float T0;
uniform float T1;
uniform float maxDist;

void main()
{
      vec4 colorBeg, colorEnd;
      if(chosenField == 1) { // temperature
          field = clamp((field - T0) / (T1 - T0), 0.0, 1.0);
          colorBeg = vec4(0, 0, 1, 0);
          colorEnd = vec4(1, 0, 0, 0);
      }
      else if(chosenField == 3) { // particle density
          field = clamp(field, 0.0, 1.0);
          colorBeg = vec4(1, 1, 1, 0);
          colorEnd = vec4(1, 0, 0, 0);
      }
      else if(chosenField == 2) { // distance field
          field = clamp(0.5 * (field + maxDist) / maxDist, 0.0, 1.0);
          colorBeg = vec4(0, 0, 1, 0);
          colorEnd = vec4(1, 0, 0, 0);
      }
      else { // phase
          field = clamp(field, 0.0, 1.0);
          colorBeg = vec4(0.1098, 0.02745, 0.62745, 0);
          colorEnd = vec4(0.8314, 0.9412, 1, 0);
      }

      gl_FragColor = (1.0 - field) * colorBeg + field * colorEnd;
}
```